Practical Lattice Basis Sampling Reduction

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des Grades
Doctor rerum naturalium (Dr. rer. nat.)

von

Dipl.-Math. Christoph Ludwig

aus Pforzheim

Referenten:	Prof. Dr. J. Buchmann (TU Darmstadt)
	Prof. Dr. C.-P. Schnorr (Universität Frankfurt a. M.)
Tag der Einreichung:	18. Oktober 2005
Tag der mündlichen Prüfung:	13. Dezember 2005

Darmstadt, 2005
Hochschulkennziffer: D 17

Wissenschaftlicher Werdegang des Verfassers[1]

10/1994 – 07/1998 und 09/1999 – 09/2000	Studium der Mathematik mit Nebenfach Informatik an der Universität Tübingen
08/1998 – 06/1999	Studium der Mathematik an der University of Oregon, Eugene, USA
12.06.1999	Graduierung von der University of Oregon (Master of Science)
27.09.2000	Abschluß der Diplom-Prüfung (Dipl.-Math.) an der Universität Tübingen
10/2000 – 09/2005	Wissenschaftlicher Mitarbeiter am Fachgebiet Theoretische Informatik (Prof. J. Buchmann), Fachbereich Informatik, Technische Universität Darmstadt

Erklärung[2]

Hiermit erkläre ich, dass ich die vorliegende Arbeit – mit Ausnahme der in ihr ausdrücklich genannten Hilfen – selbständig verfasst habe.

[1] gemäß §20 Abs. 3 der Promotionsordnung der TU Darmstadt
[2] gemäß §9 Abs. 1 der Promotionsordnung der TU Darmstadt

I would like to thank Prof. Johannes Buchmann for asking me to join his research group, for giving me the opportunity to pursue a rather broad range of topics, and for pushing me to finish this thesis. I also want to thank Prof. Claus-Peter Schnorr for evaluating the thesis and for his contributions to lattice basis reduction that I could build upon.

I would like to thank the members of the Theoretical Computer Science group in Darmstadt for their friendship and the pleasant working environment. In particular, I want to thank Marita Skrobic who manages the secretariat with competence that must not be under-appreciated. My other current and former colleagues in Darmstadt are too numerous to mention here everyone by name, but I am most grateful for all the discussions, conversations, and also the fun we shared.

Last but not least I would like to thank my grandmother, my parents, and my sister with her family. They enabled me to follow my propensities all over my education, they gave me the self-confidence to continue whenever I encountered a dry spell, and I could always count on their support in every respect.

"Tank you very much!" to all of you.

Darmstadt, December 2005 Christoph Ludwig

Zusammenfassung

Die vorgelegte Arbeit beschäftigt sich mit der Bedeutung von Sampling für die Gitterbasisreduktion und ihre Anwendungen, aufbauend auf Schnorrs *Random Sampling Reduction* Algorithmus (RSR) [Sch03]. Ferner stellt die Arbeit ein neu entwickeltes Framework für Gitterreduktionsalgorithmen vor.

Gitter sind diskrete Untergruppen des $\mathbb{R}^n$. Gitterbasisreduktion hat das Ziel, zu einem Gitter eine Basis aus möglichst kurzen Vektoren zu finden. Dies ist in der Mathematik ein seit langem betrachtetes algorithmisches Problem, das z. B. schon von Lagrange und Gauß im Kontext der quadratischen Formen behandelt wurde. Lange Zeit blieb Gitterbasisreduktion ein Spezialthema der Zahlentheorie; doch nachdem Lenstra, Lenstra und Lovász [LLL82] den polynomiellen LLL-Algorithmus vorgestellt hatten, fanden sich rasch zahlreiche Anwendungen in der ganzzahligen Optimierung, der Kodierungstheorie und der Kryptographie.

Die klassischen Probleme in der Gittertheorie sind NP-hard; mit Hilfe von LLL kann man approximative Lösungen bestimmen, wobei der Approximationsfaktor exponentiell in der Gitterdimension steigt. In den vergangenen 20 Jahren wurden deshalb weitere Reduktionsalgorithmen entwickelt, die einen verbesserten Approximationsfaktor gegen längere Laufzeiten eintauschen. Der bekannteste dieser Algorithmen ist die BKZ-Reduktion [Sch87, SE94], die – für einen vom Anwender gewählten Parameter k – den Approximationsfaktor um den Faktor $1/k$ im Exponenten verbessert, allerdings um den Preis eines schlechtestenfalls in k superexponentiellen zusätzlichen Rechenaufwandes.

Schnorr hat 2003 mit dem RSR-Algorithmus eine neue Technik zur Gitterbasisreduktion vorgestellt, die Algorithmen vom LLL-Typ mit der erschöpfenden Suche in einer Menge von Gittervektoren kombiniert, die mit nicht vernachlässigbarer Wahrscheinlichkeit einen kurzen Gittervektor enthält. Schnorr kam zu dem Ergebnis, dass RSR die bis dahin effizientesten Algorithmen um eine vierte Wurzel verbessert. Allerdings basiert RSR auf zwei Annahmen hinsichtlich der Gram-Schmidt-Zerlegung LLL-reduzierter Basen und des statistischen Verhaltens der Koordinaten von Gitterpunkten bezüglich dieser Gram-Schmidt-Zerlegung. Aus algorithmischer Sicht kommt dabei der sog. Geometric Series Assumption (GSA) besondere Bedeutung zu. In der Praxis können wir bestenfalls erwarten, dass LLL-reduzierte Basen diesen Annahmen in einem approximativen Sinne entsprechen, was Fragen hinsichtlich der Praktikabilität von RSR aufwirft.

Wir geben in Kapitel 2 eine hinreichende Bedingung an, die leicht numerisch überprüft werden kann und die garantiert, dass Schnorrs Analyse für eine Gitterbasis

zutrifft, auch wenn diese (GSA) nicht exakt erfüllt. Wir zeigen aber auch, dass schon wenige RSR-Iterationen die (GSA)-Eigenschaft dermaßen stören, dass wir die von Schnorr gefolgerten asymptotischen Verbesserungen in der Praxis nicht erwarten können.

Trotz alledem ist es möglich, RSR so zu modifizieren, dass der Algorithmus auch dann wohldefiniert ist, wenn Schnorrs Annahmen nicht zutreffen. Wir beschreiben in Kapitel 3 eine derartige Variante und entwickeln zwei Teilalgorithmen, die entscheiden, ob eine Suche nach kurzen Vektoren in dem jeweiligen Suchraum eine hinreichende Chance auf Erfolg hat. Der erste Algorithmus setzt die in Schnorrs RSR-Analyse entwickelte Idee algorithmisch um und berechnet eine recht grobe, meist zu pessimistische Abschätzung der Erfolgswahrscheinlichkeit. Der zweite, etwas aufwändigere Algorithmus implementiert eine direkte Auswertung der Wahrscheinlichkeitsfunktion für die Länge der Vektoren im Suchraum, basierend auf dem Faltungssatz, und ergibt genauere Abschätzungen. Ferner stellen wir zwei Varianten der Sampling-Reduktion vor, die zum einen darauf abzielen, die in der Sampling-Phase gefundenen Vektoren unterdurchschnittlicher Länge auszunutzen, und die zum anderen der Störung der (GSA)-Eigenschaft bzw. deren Konsequenzen gegensteuern.

Quantencomputer werden – wenn sie denn eines Tages zur Verfügung stehen werden – in der Lage sein, Objekte, die ein gegebenes Prädikat erfüllen, in ungeordneten Mengen mit sublinearem Aufwand zu finden. Wir machen uns dies in Kapitel 4 zu Nutze, um eine Quanten-Sampling-Reduktion zu entwickeln, die im Vergleich zur klassischen Variante asymptotisch um eine Quadratwurzel schneller ist.

Die experimentelle Auswertung der verschiedenen Algorithmen zur Sampling-Reduktion machte ein Werkzeug erforderlich, das die rasche Implementierung verschiedener algorithmischer Varianten ermöglicht und eine geeignete Testumgebung zur Verfügung stellt. Wir beschreiben in Kapitel 5 das C++-Framework LaRed und die dazugehörige Softwarebibliothek, die wir zu diesem Zweck entwickelten. Wir stellen auch deren Anbindung an die Skriptsprache Python vor, mit deren Hilfe die interaktive Kontrolle der Experimente bis hin zu Modifikationen der Algorithmen zur Laufzeit möglich werden.

Wir schließen die Dissertation in Kapitel 6 mit Berichten über das Verhalten von Sampling-Reduktion bei der Anwendung auf Gitterbasen ab, die sich aus kryptographischen Anwendungen ergeben: Öffentliche Schlüssel aus Micciancios Variante des GGH Kryproverfahrens, NTRU-Gitter sowie Knapsack-Gitter. Wir stellten bei unseren Experimenten fest, dass Sampling-Reduktion ermöglicht, einen deutlich kleineren BKZ-Parameter zu wählen als bei ausschließlicher Reduktion mit BKZ. Zwar war die Gesamtlaufzeit der Reduktion trotz des kleineren BKZ-Parameters häufig länger als bei alleiniger Anwendung von BKZ, aber der Großteil dieser Rechenzeit wurde auf das Sampling verwandt. Weil dieses ohne weiteres auf zahlreiche Knoten in einem Netzwerk aus günstiger Hardware verteilt werden kann, scheint Sampling-Reduktion einem Angreifer dennoch einen Vorteil zu bieten.

Abstract

This thesis studies the impact of sampling on lattice basis reduction and its applications, advancing on the *Random Sampling Reduction* (RSR) algorithm proposed by Schnorr [Sch03], and introduces a framework for lattice reduction by sampling algorithms.

A lattice is a discrete subgroup of $\mathbb{R}^n$. Lattice basis reduction aims to efficiently compute lattice bases that consist of vectors as short as possible. This algorithmic problem goes back to Lagrange and Gauß in the context of the theory of quadratic forms. When the polynomial LLL algorithm [LLL82] became available, lattice reduction stopped being a topic for specialists in number theory only; it was soon applied in integer linear programming, coding theory, and cryptography.

The most important problems in lattice theory are NP-hard; the solutions computed by LLL are only approximations up to an exponential factor. Over the last twenty years, many algorithms were proposed that advance on LLL, trading better approximations for longer computing times. The most important one is the BKZ algorithm [Sch87, SE94] that improves the exponent of the approximation factor by $1/k$ for some user defined parameter k at the cost of additional worst case runtime super-exponential in k.

Recently, Schnorr [Sch03] proposed RSR, a new lattice reduction technique that combines LLL-like algorithms with the exhaustive search of a set of lattice points that is likely to contain short vectors. He concluded that his new algorithm improves on the previous most efficient algorithm by a fourth root. However, RSR depends on two assumptions on the Gram-Schmidt decomposition of LLL reduced bases and the coordinates of lattice points with respect to these Gram-Schmidt decompositions where, from an algorithmic point of view, the so called Geometric Series Assumption (GSA) is particularly crucial. In practice, we can, at best, expect LLL reduced lattice bases to satisfy these assumptions in an approximate sense, casting doubt on the practicality of RSR.

We give in Chapter 2 a sufficient condition that can be numerically evaluated and that guarantees a lattice basis approximates (GSA) well enough for Schnorr's analysis to be valid. We show that, even if the input basis comes sufficiently close to (GSA), few iterations of Sampling Reduction will disrupt this property, whence we cannot expect Sampling Reduction to achieve in practice the asymptotic improvement that follows under Schnorr's assumptions.

Nevertheless, Sampling Reduction can be modified into a practical algorithm that is well defined without (GSA). We describe in Chapter 3 such a variant and develop

two subalgorithms that determine whether it is sufficiently likely that the sampling's search space contains a short vector. The first one implements the approach Schnorr took in his analysis of RSR and gives a quite rough estimate of the probability. The second, algorithmically more complex one is based on a more direct evaluation of the probability function of the length of the sampled vectors by means of the convolution theorem and gives a more accurate estimate. We propose two further Sampling Reduction algorithms that aim to take advantage of the relatively short vectors found while sampling and that allow, to some degree, to work around the disruption of the (GSA) property.

If quantum computers become available, they will be able to search unordered sets in sublinear time, unlike classical computers. We use this in Chapter 4 to develop a Quantum Sampling Reduction that, asymptotically, speeds the classical reduction up by a square root.

The empirical analysis of Sampling Reduction algorithms required a tool that eases the implementation of various algorithmic variants and provides a testbed. We describe in Chapter 5 the C++ framework LaRed and its accompanying library we developed for this purpose as well as its binding to the scripting language Python that allows the interactive control of reductions and even the modification of parts of the algorithms at runtime.

We conclude this thesis in Chapter 6 with reports on the performance of our Sampling Reduction algorithms if applied to lattice basess that stem from cryptographic applications: Public keys in Micciancio's variant of the GGH crypto-scheme, NTRU lattices, and knapsack lattices. We found that the BKZ parameter required for these applications is significantly smaller with Sampling Reduction than with BKZ alone. Even though the total runtime of Sampling Reduction was often longer than with BKZ, most of it was spent on sampling. Since sampling can easily be distributed, Sampling Reduction is nevertheless likely to offer an attacker an advantage.

Contents

Zusammenfassung **v**

Abstract **vii**

1. Introduction **1**
1.1. Notation . . . 1
1.2. Preliminaries and Previous Work . . . 2

2. Schnorr's Random Sampling Reduction **9**
2.1. The RSR Algorithm . . . 9
2.2. The Geometric Series Assumption . . . 10
2.3. The Candidate Set . . . 11
2.4. The Randomness Assumption . . . 14
2.5. Analysis of RSR . . . 15
2.6. Problems with RSR in Practice . . . 20

3. Practical Sampling Reduction **25**
3.1. Simple Sampling Reduction . . . 25
3.1.1. The Simple Sampling Reduction Algorithm . . . 25
3.1.2. Practical Behavior of SSR . . . 27
3.2. Trivial CSSS . . . 28
3.3. CSSS Following Schnorr's Approach . . . 29
3.3.1. The expected length of $\mathbf{v}$. . . . 29
3.3.2. The $\mathsf{CSSS}_{\mathsf{event}}$ implementation. . . . 30
3.4. CSSS by Convolution . . . 32
3.4.1. Preliminaries . . . 32
3.4.2. The Probability Function of $\|\mathbf{v}\|^2$ . . . 34
3.4.3. The $\mathsf{CSSS}_{\mathsf{Fourier}}$ Algorithm . . . 37
3.5. Pool Sampling Reduction . . . 41
3.5.1. The PoolSR Algorithm and Randomized Acceptors . . . 42
3.5.2. Empirical Behavior of PoolSR . . . 44
3.6. Short Projection Sampling Reduction . . . 47
3.6.1. The ShortProjectionSR Algorithm . . . 47
3.6.2. Empirical Behavior of ShortProjectionSR . . . 49
3.7. Further Generalizations . . . 53
3.7.1. Search Space Generalization . . . 54

3.7.2. Distributed Sampling . . . 57
3.7.3. Gram Matrices . . . 57

4. Quantum Lattice Reduction **59**
4.1. Grover's Quantum Search . . . 59
4.2. The Quantum Search Reduction Algorithm . . . 63
4.3. QSR vs. Classical Algorithms . . . 65

5. The LaRed Framework **69**
5.1. LaRed . . . 69
5.1.1. Design Goals and Requirements . . . 69
5.1.2. Design Choices . . . 71
5.1.3. Implementation . . . 75
5.2. laredpy . . . 82
5.2.1. The LaRed Binding to Python . . . 83
5.2.2. Evaluating Reduction Algorithms with laredpy . . . 84

6. Cryptographic Applications **89**
6.1. Micciancio's GGH Variant . . . 89
6.1.1. Micciancio's Cryptosystem . . . 90
6.1.2. Lattice Basis Reduction Attacks . . . 92
6.2. NTRU . . . 97
6.2.1. NTRU Lattices . . . 97
6.2.2. Attacks by Sampling Reduction . . . 99
6.3. Knapsack Lattices . . . 102
6.3.1. Knapsack Lattice Bases . . . 102
6.3.2. Sampling Reduction of Knapsack Lattice Bases . . . 104

A. Sampling Results **107**
A.1. Micciancio Attack Keys . . . 108
A.2. Micciancio Embedding Attacks . . . 119
A.3. NTRU Lattices . . . 123
A.4. Knapsack Lattices . . . 128

Bibliography **135**

Chapter 1.

Introduction

This thesis studies the impact of sampling on lattice basis reduction and its applications, advancing on the *Random Sampling Reduction* (RSR) algorithm proposed by Schnorr [Sch03], and introduces a framework for lattice reduction by sampling algorithms.

In this chapter we describe the topic of lattice basis reduction, report previous results, and introduce the notation and mathematical preliminaries used in this thesis. In Chapter 2 we outline Schnorr's algorithm RSR and its analysis. We discuss how two assumptions that Schnorr made affect the practicability of RSR. We develop in Chapter 3 variants of Sampling Reduction that are well defined and feasible even if Schnorr's assumptions do not hold. For this we propose strategies to estimate the probability to find a sufficiently short lattice vector in a given sample space and evaluate modifications of the basic algorithm that aim to overcome the effects that often make Sampling Reduction stall after few iterations. We propose in Chapter 4 a quantum algorithm variant of Sampling Reduction based on Grovers quantum search algorithm that asymptotically gives a speed up. In Chapter 5 we describe a framework that was developed to aid the implementation of and experimenting with various Sampling Reduction Algorithms, both in batch-mode and in an interactive setting. We finally show in Chapter 6 how Sampling Reduction performs in some cryptographic applications.

1.1. Notation

Unless we explicitly say otherwise, we use the following conventions and notations.

The symbols $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}$ denote the set of natural, integral, rational, real, and complex numbers, respectively, where we consider $0 \in \mathbb{N}$. If A is any real set, then $A_+ := \{\, a \in A \mid a > 0 \,\}$. $\lfloor \cdot \rfloor$ stands for "rounding down", i. e.,

$$\lfloor x \rfloor := \max\{\, n \in \mathbb{Z} \mid n \leq x \,\}$$

for $x \in \mathbb{R}$. Similarly, $\lceil \cdot \rceil$ means "rounding up"

$$\lceil x \rceil := \min\{\, n \in \mathbb{Z} \mid n \geq x \,\}$$

and $\lceil x \rfloor := \lceil x - 0.5 \rceil$ is rounding to the closest integer.

$M^n := M \times \cdots \times M$ is the n-fold cartesian product of the set M with itself. $R^{m\times n}$ is the set of matrices $\mathsf{A} = (a_{i,j})$ over the ring R with row index $i \in \{1, \ldots, m\}$ and column index $j \in \{1, \ldots, n\}$. Matrices operate on vectors by left multiplication, i. e., we view vectors $\mathbf{v} \in R^n$ as column vectors. $[\mathbf{v}_1, \ldots, \mathbf{v}_n]$ is the matrix in $R^{m\times n}$ with columns $\mathbf{v}_1, \ldots, \mathbf{v}_n \in R^m$. Transposition of vectors and matrices is denoted by a superscript t.

For any complex number $z = x + \imath y$, its complex conjugate is $\overline{z} := x - \imath y$. The complex vector space $\mathbb{C}^n$ carries the euclidean metric, i. e., the scalar product

$$\langle \mathbf{x}, \mathbf{y} \rangle = \overline{\mathbf{x}}^t \mathbf{y} = \sum_{j=1}^{n} \overline{x_j} y_j \qquad \text{for } \mathbf{x} = (x_1, \ldots, x_n)^t, \mathbf{y} = (y_1, \ldots, y_n)^t \in \mathbb{C}^n$$

and the corresponding euclidean norm $\|\mathbf{x}\| = \langle \mathbf{x}, \mathbf{x} \rangle^{1/2}$. The metric on $\mathbb{Z}^n$, $\mathbb{Q}^n$, $\mathbb{R}^n$, and their subsets is induced by their embedding in $\mathbb{C}^n$.

Let $V \subseteq \mathbb{C}^n$ be an R-module, $R \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}, \mathbb{C}\}$, and let $W \subseteq V$ be a subset. Then $\operatorname{lin}_R W = \{ r_1\mathbf{w}_1 + \cdots + r_k\mathbf{w}_k \mid k \in \mathbb{N}, r_j \in R, \mathbf{w}_j \in W \}$ is the R-module spanned by W. The orthogonal space of W is $W^\perp = \{ \mathbf{v} \in V \mid \langle \mathbf{v}, \mathbf{w} \rangle = 0 \text{ for all } \mathbf{w} \in W \}$.

$C(\mathbb{R})$ is the set of continuous functions on $\mathbb{R}$, $C^1(\mathbb{R})$ is the set of continously differentiable functions and so forth.

We describe the asymptotic behavior of real valued functions f, g by the Landau notation: $f \in O(g)$ means g is an asymptotic upper bound for f (in other words, $\limsup_{n\to\infty} |f(n)/g(n)|$ is finite). If g grows faster than f (i. e., $\lim_{n\to\infty} |f(n)/g(n)| = 0$), then we write $f \in o(g)$. $f \in \Theta(g)$ implies that f and g show the same asymptotic behavior (i. e., $f \in O(g)$ and $g \in O(f)$). And finally, $f \in \Omega(g)$ if and only if g is an asymptotic lower bound for f (i. e., $\liminf_{n\to\infty} |f(n)/g(n)| > 0$).

1.2. Preliminaries and Previous Work

Below we define lattices and state some elementary facts about them. We introduce some classical lattice problems and mention previous work in the context of lattice basis reduction. For proofs and further details we refer to, e. g., [MG02].

Definition 1. *A* lattice *L is a discrete additive subgroup of $\mathbb{R}^d$, $d \in \mathbb{N}$. L is called* integral *if and only if $L \subseteq \mathbb{Z}^d$.*

Lattices turn out to be free unitary $\mathbb{Z}$-modules of finite rank, i. e., for any lattice L there is a $\mathbb{Z}$-linearly independent set $B = \{ \mathbf{b}_1, \ldots, \mathbf{b}_n \}$, $n \in \mathbb{N}$, such that $L = \operatorname{lin}_{\mathbb{Z}} B$ and the cardinality of any such set is invariant. Since L is discrete, B is necessarily $\mathbb{R}$-linearly independent as well.

Definition 2. *Let $L \subset \mathbb{R}^d$ be a lattice. $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{d\times n}$ is an* ordered basis *of $L = L(\mathsf{B})$ if and only if $\{ \mathbf{b}_1, \ldots, \mathbf{b}_n \}$ is $\mathbb{R}$-linearly independent and spans the $\mathbb{Z}$-module L. n is then called the* dimension *of L. If $n = d$, then L is said to be* fully dimensional.

We consider ordered bases of lattices only whence we drop in the following the adjunct "ordered". Clearly, a matrix $\mathsf{B} \in \mathbb{R}^{d\times n}$ is a lattice basis if and only if B has full column rank.

Lemma 1. *Two lattice bases* $\mathsf{B}, \mathsf{B}' \in \mathbb{R}^{d\times n}$ *generate the same lattice if and only if there is a unimodular matrix* $\mathsf{U} \in \mathbb{Z}^{n\times n}$ *(i. e.,* $\det(\mathsf{U}) = \pm 1$*) such that* $\mathsf{B}' = \mathsf{BU}$.

Definition 3. *Let* L *be a lattice with basis* $\mathsf{B} \in \mathbb{R}^{d\times n}$. *The* determinant *of* L *is* $\det(L) := |\det(\mathsf{B}^t\mathsf{B})|^{1/2}$.

The determinant is a lattice invariant by Lemma 1. If L is fully dimensional, then $\det(L) = |\det(\mathsf{B})|$. Therefore, $\det(L)$ can be interpreted as the volume of the parallelepiped spanned by the columns of B:

$$\det(L) = \mathrm{Vol}(\{\, \mathsf{B}\mathbf{x} \mid \mathbf{x} \in [0,1)^n \,\})$$

Definition 4. *Let* $\mathsf{B} \in \mathbb{R}^{d\times n}$ *be a basis of the lattice* L. *The* orthogonality defect *of* B *is defined by*

$$\mathrm{odef}(\mathsf{B}) = \frac{\prod_{j=1}^{n} \|\mathbf{b}_j\|}{\det(L)}\,.$$

We have $\mathrm{odef}(\mathsf{B}) \geq 1$ for any lattice basis B with equality if and only if the columns of B are pairwise orthogonal.

Definition 5. *Let* L *be an* n*-dimensional lattice and* $j \in \{\,1, \ldots, n\,\}$. *The real number*

$$\lambda_j(L) := \min\{\, \max\{\, \|\mathbf{v}_1\|, \ldots, \|\mathbf{v}_j\| \,\} \mid \{\, \mathbf{v}_1, \ldots, \mathbf{v}_j \,\} \subset L \text{ linearly independent} \}$$

is named the jth successive minimum *of* L.

In particular, $\lambda_1(L)$ is the euclidean length of the shortest nontrivial vectors in L. That poses the question how to construct such vectors. In computational lattice theory, we need to represent the lattice bases in finite space and the runtime of our algorithms is a function of the input basis size. In the following, we will therefore mostly consider integral lattices. The results can be carried over to lattices in $\mathbb{Q}^d$ by scaling with the least common denominator. Irrational lattices need to be approximated [Buc94].

Problem 1 (Approximate Shortest Vector Problem, α-SVP). *Let* $\alpha \geq 1$. *Given a basis* $\mathsf{B} \in \mathbb{Z}^{d\times n}$ *of the lattice* $L = L(\mathsf{B})$, *find a lattice vector* $\mathbf{v} \in L$ *with* $\|\mathbf{v}\| \leq \alpha\lambda_1(\mathsf{B})$.

It was not long ago that 1-SVP was proved to be NP-hard under randomized reductions [Ajt98]. In fact, even the approximate α-SVP for $\alpha < \sqrt{2}$ is NP-hard [Mic01c]. On the other hand, α-SVP cannot be NP-hard anymore for $\alpha \in \Omega(\sqrt{n}/\log n)$ unless the complexity class NP is contained in coAM, which is believed to be unlikely [GG00].

In contrast to the α-SVP, that asks for a single short vector only, the Quasi Orthogonal Basis Problem is to find a basis of overall minimal length:

Problem 2 (Approximate Quasi Orthogonal Basis Problem, α-QOB). *Let $\alpha \geq 1$. Given a basis $\mathsf{B} \in \mathbb{Z}^{d\times n}$ of the lattice $L = L(\mathsf{B})$, find a basis $\mathsf{B}' = [\mathbf{b}'_1, \ldots, \mathbf{b}'_n] \in \mathbb{Z}^{d\times n}$ of L such that*

$$\prod_{j=1}^{n} \|\mathbf{b}'_j\| \leq \alpha \min\left\{ \prod_{j=1}^{n} \|\mathbf{a}_j\| \,\middle|\, [\mathbf{a}_1, \ldots, \mathbf{a}_n] \text{ is a basis of } L \right\}$$

or, equivalently, $\operatorname{odef}(\mathsf{B}') \leq \alpha \min\{ \operatorname{odef}(\mathsf{A}) \mid \mathsf{A} \text{ is a basis of } L \}$.

QOB is sometimes named *Smallest Basis Problem*. Other authors refer by the Smallest Basis Problem to the task to minimize $\max\{ \|\mathbf{b}_1\|, \ldots, \|\mathbf{b}_n\| \}$ (e. g., [NS01]) whence we avoid this name. QOB is mentioned in the literature (e. g., [GGH97]), but we are not aware of published complexity results. Wetzel [Wet98] states that 1-QOB is NP-hard according to an unpublished result by Lovász.

The NP-hardness of 1-SVP had been conjectured for some time since the related Closest Vector Problem was already known to be NP-hard [EB81].

Problem 3 (Approximate Closest Vector Problem, α-CVP). *Let $\alpha \geq 1$. Given a basis $\mathsf{B} \in \mathbb{Z}^{d\times n}$ of the lattice $L = L(\mathsf{B})$ and $\mathbf{x} \in \mathbb{Z}^d$, find a lattice vector $\mathbf{v} \in L$ subject to $\|\mathbf{v} - \mathbf{x}\| \leq \alpha \min\{ \|\mathbf{w} - \mathbf{x}\| \mid \mathbf{w} \in L \}$.*

α-CVP stays NP-hard if $\alpha \leq n^{c/\log\log n}$ for some $c > 0$ [ABSS97, DKRS03].

If we refer to SVP, QOB, or CVP unqualified, then we mean the non-approximate problem with $\alpha = 1$, respectively. The complexity results mentioned above are about worst-case complexity. In fact, the hardness of α-SVP strongly depends on the available basis. In practice, the same holds for α-CVP despite a result by Micciancio that arbitrary preprocessing of the lattice basis does not change the NP-hardness of CVP [Mic01a]. This gives rise to so called *lattice basis reduction* that aims to compute "good" bases. In general, the shorter and the less skewed the base vectors, the better is the basis.

Theorem 2. *Let $\mathsf{B} \in \mathbb{R}^{d\times n}$ be a lattice basis. The decomposition $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$ of B into a matrix $\hat{\mathsf{B}} = [\hat{\mathbf{b}}_1, \ldots, \hat{\mathbf{b}}_n] \in \mathbb{R}^{d\times n}$ with pairwise orthogonal columns and a unit upper triangular matrix $\mathsf{R} = (r_{i,j}) \in \mathbb{R}^{n\times n}$ can be efficiently computed by the* Gram-Schmidt method*:*

$$\begin{aligned}
\hat{\mathbf{b}}_1 &= \mathbf{b}_1 \\
\hat{\mathbf{b}}_j &= \mathbf{b}_j - \sum_{l=1}^{j-1} r_{l,j}\hat{\mathbf{b}}_l && \text{for } 1 < j \leq n \\
r_{l,j} &= \frac{\langle \hat{\mathbf{b}}_l, \mathbf{b}_j \rangle}{\|\hat{\mathbf{b}}_l\|^2} && \text{for } 1 \leq l < j \leq n
\end{aligned}$$

For numerical reasons, QR algorithms based on Householder or fast Givens rotations [GL96] are preferred in practice, but they result in the same decomposition. Lattice basis reduction exploits two properties of the Gram-Schmidt decomposition:

Lemma 3. *Let* $\mathsf{B} \in \mathbb{R}^{d\times n}$ *a basis of the lattice* L *with Gram-Schmidt decomposition* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$, $\hat{\mathsf{B}} = [\hat{\mathbf{b}}_1, \ldots, \hat{\mathbf{b}}_n]$. *Then*

$$\det(L) = \prod_{j=1}^{n} \|\hat{\mathbf{b}}_j\| \qquad \textit{and} \qquad \lambda_1(L) \geq \min\{\, \|\hat{\mathbf{b}}_1\|, \ldots, \|\hat{\mathbf{b}}_n\| \,\}\,.$$

The first lattice basis reduction algorithm goes back to Gauss [Gau01]. His reduction algorithm solves the SVP in 2-dimensional lattices. Algorithmic lattice theory attracted interest again when Lenstra, Lenstra, and Lovász proposed their renowned LLL algorithm [LLL82].

Definition 6. *Let* $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d\times n}$ *be a lattice basis with Gram-Schmidt decomposition* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$, $\hat{\mathsf{B}} = [\hat{\mathbf{b}}_1, \ldots, \hat{\mathbf{b}}_n]$, $\mathsf{R} = (r_{i,j})$, *and let* $\delta \in (1/4, 1]$. B *is said to be* δ-LLL-*reduced if and only if*

$$\begin{aligned} |r_{i,j}| &\leq 1/2 && \textit{for } 1 \leq i < j \leq n \textit{ and} \\ \|\hat{\mathbf{b}}_{j+1}\|^2 &\geq (\delta - r_{j,j+1}^2)\|\hat{\mathbf{b}}_j\|^2 && \textit{for } 1 \leq j < n. \end{aligned}$$

If $\log_2 \|\mathbf{b}_j\| \in O(n)$ for all input basis vectors $\mathbf{b}_j$, then the LLL algorithm returns in $O(dn^4)$ arithmetic steps a δ-LLL-reduced basis provided $1/4 < \delta < 1$. These arithmetic steps are performed on integers of bit length $O(n^2)$. The following theorem shows that the LLL algorithm computes approximate solutions to the SVP and QOB.

Theorem 4. *Let* $\delta \in (1/4, 1]$ *and let* $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d\times n}$ *be a* δ-LLL-*reduced basis of the lattice* L *with Gram-Schmidt decomposition* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$, $\hat{\mathsf{B}} = [\hat{\mathbf{b}}_1, \ldots, \hat{\mathbf{b}}_n]$. *Set* $\alpha = (\delta - 1/4)^{-1}$. *Then*

$$\|\mathbf{b}_1\| \leq \alpha^{(n-1)/4} \det(L)^{1/n} \qquad\qquad \prod_{l=1}^{n} \|\mathbf{b}_l\| \leq \alpha^{n(n-1)/4} \det(L)$$

and, for $1 \leq i \leq j \leq n$,

$$\|\mathbf{b}_i\| \leq \alpha^{(j-1)/2}\|\hat{\mathbf{b}}_j\| \qquad\qquad \alpha^{(1-i)/2}\lambda_i(L) \leq \|\mathbf{b}_i\| \leq \alpha^{(n-1)/2}\lambda_i(L)\,.$$

The LLL algorithm operates on the rational numbers $\|\hat{\mathbf{b}}_j\|^2$ and $r_{i,j}$. In practice, these operations are expensive in exact arithmetic, whence one uses almost always a variant due to Schnorr and Euchner [SE94] that operates on floating point numbers but avoids excessive floating point errors, reducing the cost of LLL to $O(dn^4)$ arithmetic steps on integers of bit length $O(n)$.

Wetzel proposed several algorithmic variants of LLL like iterative or modular reductions and described their empirical behavior in [Wet98]. There are also parallel reduction algorithms in, e. g., [Hec95, Wet98]. They are mostly for machines where the computing nodes share their memory or are at least tightly coupled by a communication network.

Definition 7. *Let* $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{d \times n}$ *be a basis of the lattice* L *with Gram-Schmidt decomposition* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$, $\hat{\mathsf{B}} = [\hat{\mathbf{b}}_1, \ldots, \hat{\mathbf{b}}_n]$. *For* $j \in \{1, \ldots, n\}$ *set*

$$\mathsf{P}_j = [\hat{\mathbf{b}}_1/\|\hat{\mathbf{b}}_1\|, \ldots, \hat{\mathbf{b}}_{j-1}/\|\hat{\mathbf{b}}_{j-1}\|] \in \mathbb{R}^{d \times (j-1)}.$$

Then

$$\pi_j : \mathbb{R}^d \to \mathbb{R}^d : \mathbf{x} \mapsto \mathbf{x} - \mathsf{P}_j \mathsf{P}_j^t \mathbf{x}$$

is the orthogonal projection onto the orthogonal space $W_j := \{\mathbf{b}_1, \ldots, \mathbf{b}_{j-1}\}^\perp$. *We denote the projection of* L *onto* W_j *by* $L_j(\mathsf{B}) := \{\pi_j(\mathbf{v}) \mid \mathbf{v} \in L\}$.

Note that π_j depends on the basis of L and that π_1 is the identity. $L_j(\mathsf{B})$ is also a lattice.

Definition 8. *A lattice basis* $\mathsf{B} \in \mathbb{R}^{d \times n}$ *with Gram-Schmidt decomposition* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$, $\hat{\mathsf{B}} = [\hat{\mathbf{b}}_1, \ldots, \hat{\mathbf{b}}_n]$, $\mathsf{R} = (r_{i,j})$, *is called* Korkine-Zolotarev reduced *if and only if*

$$\begin{aligned} &|r_{i,j}| \leq 1/2 && \text{for } 1 \leq i < j \leq n \text{ and} \\ &\|\hat{\mathbf{b}}_j\| = \lambda_1(L_j(\mathsf{B})) && \text{for } 1 \leq j < n. \end{aligned}$$

Clearly, the first vector in a Korkine-Zolotarev reduced basis is a solution to the SVP in the generated lattice. But the remaining base vectors are also very short:

Theorem 5. *Let* $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{d \times n}$ *be a Korkine-Zolotarev reduced basis of the lattice* L. *Then, for all* $j \in \{1, \ldots, n\}$,

$$\frac{4}{3+j}\lambda_j(L)^2 \leq \|\mathbf{b}_j\|^2 \leq \frac{3+j}{4}\lambda_j(L)^2.$$

There are algorithms to compute Korkine-Zolotarev reduced bases [Kan83, Kan87, LLS90], but their runtime is exponential in the lattice dimension n. Therefore, Schnorr proposed a parameterized notion of reducedness that forms a hierarchy with Korkine-Zolotarev at one and LLL at the other end and analyzed the approximation factor guaranteed by this reduction [Sch87, Sch94].

Definition 9. *A lattice basis* $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{d \times n}$ *is called block Korkine-Zolotarev reduced with block size* $\beta \in \{2, \ldots, n\}$ *(or, short,* β-BKZ *reduced) if and only if* $\mathsf{B}_j = [\pi_j(\mathbf{b}_j), \ldots, \pi_j(\mathbf{b}_{\min\{j+\beta-1, n\}})]$ *is Korkine-Zolotarev reduced for all* $j \in \{1, \ldots, n\}$.

Theorem 6. *Let* $\mathsf{B} \in \mathbb{R}^{d \times n}$ *be a* β*-BKZ reduced basis of the lattice* L, $\beta \in \{2, \ldots, n\}$ *and* $\gamma_\beta := \sup\{\lambda_1(L')^2/\det(L')^{2/\beta} \mid L'$ *is a* β*-dimensional lattice*$\}$ *the* β^{th} *Hermite constant. Then* B *is also* 1-LLL *reduced and, for all* $j \in \{1, \ldots, n\}$,

$$\|\mathbf{b}_j\|^2 \leq \gamma_\beta^{2(n-1)/(\beta-1)} \frac{3+j}{4}\lambda_j(L)^2 \leq \left(\frac{2}{3}\beta\right)^{2(n-1)/(\beta-1)} \frac{3+j}{4}\lambda_j(L)^2.$$

Schnorr gives in [Sch87] a (semi) $2k$-reduction that computes, for a fully dimensional lattice, in $O(n^3 k^{k+o(k)} + n^4)$ time an (almost) $2k$-BKZ reduced basis. In practice, however, one uses the (δ, β)-BKZ reduction proposed by Schnorr and Euchner [SE94] that relaxes the reduction such that $\delta \|\hat{\mathbf{b}}_j\|^2 \leq \lambda_1(\pi_j(L(\mathbf{b}_j, \ldots, \mathbf{b}_{\min\{j+\beta-1,n\}}))$ for $\delta \in (1/4, 1)$. There is no proven polynomial time bound for (δ, β)-BKZ, but in practice its worst-case runtime seems to be similar to $2k$-reduction.

The nearest plane algorithm by Babai [Bab86] computes in polynomial time a solution to $2^{n/2}$-CVP if used with LLL. The approximation can be improved if the algorithm is used in combination with Schnorr's algorithms for BKZ reduction. [AEVZ02] contains a survey on algorithms for solving CVP.

Koy and Schnorr extended Schnorr's hierarchy below LLL by (strong) segment-LLL [KS01a, KS01b, KS02]. Strong segment-LLL achieves the same SVP approximation factor as LLL but runs in only $O(n^3 \log n)$ arithmetic steps.

Another LLL-type reduction is Koy's primal-dual reduction [Koy04] that achieves the SVP approximation factor $(k/6)^{n/k}$ in $O(n^3 k^{k/2+o(k)} + n^4)$ time.

Ajtai, Kumar, and Sivakumar [AKS01] developed a sieve method to solve SVP that runs in $2^{O(n)}$ time. Unfortunately, their algorithm is not practical since the O-constant is about 30.

Schnorr [Sch03] proposed a lattice basis reduction named Random Sampling Reduction (RSR) that iteratively combines LLL with an exhaustive search to approximate SVP up to $(k/6)^{n/2k}$. He concludes that his method speeds up the fastest previous reduction by a fourth root; however, his algorithm and analysis rely on two assumptions. We will discuss in the following chapters how RSR can be turned into a practical algorithm even though the two assumptions do not necessarily hold.

Schnorr also proposes variations of RSR that exploit the birthday paradox to speed up the search for a short vector. Since the space needed for these variants is exponential in the sampling parameter k, they seem not practical, though.

Chapter 2.

Schnorr's Random Sampling Reduction

In 2003, Schnorr [Sch03] proposed a novel Random Sampling Reduction (RSR) algorithm. RSR repeatedly samples lattice points from a set of likely short candidate vectors and subsequently LLL reduces the generating system formed by the previous basis and a sampled short vector. Schnorr analyzed the runtime and the approximation factor of RSR and concluded under two additional assumptions that RSR outperforms all previous algorithms by a fourth root.

In the following, we state the algorithm, explain the assumptions required by RSR, and describe the set of candidate vectors. After the groundwork is laid, we can give an analysis of RSR in Sect. 2.5 that results in slightly stricter bounds and is more detailed than [Sch03]. In practice, we cannot expect the input lattice bases to strictly support the assumptions (GSA) and (RA) that the analysis is based on and that are stated in Sect. 2.2 and 2.4. We discuss in Sect. 2.6 the impact these assumptions have on RSR in practice and give a sufficient, easily verifiable condition that can replace (GSA) in Schnorr's analysis.

2.1. The RSR Algorithm

Schnorr's *Random Sampling Reduction* algorithm RSR is given in Alg. 1 on the following page. Note that the correctness of the algorithm as stated requires both the Geometric Series Assumption (GSA, Sect. 2.2) and the Randomness Assumption (RA, Sect. 2.4). Here we sketch how the algorithm works; a more detailed analysis is deferred to Sect. 2.5.

RSR operates exclusively on LLL reduced bases. In practice, one will most likely prefer the stronger BKZ reduction to LLL. But there is no guaranteed runtime bound for BKZ whence the analysis assumes that the initial basis reduction and every update is done with LLL.

The algorithm takes advantage of two facts: LLL needs to compute the Gram-Schmidt coefficient matrix R of the reduced basis anyway, so we can output it just as well. Second, it is straightforward to modify the original LLL algorithm in such a way that the input can be any finite generating system of the lattice. The algorithm detects linear dependencies and removes the corresponding vectors from the system. Therefore, the output of LLL is a basis even if the input generating system is linear dependent.

Algorithm 1 RSR

Input:
- basis $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d \times n}$ of lattice L
- RSR parameter k such that $24 \leq k$ and $3k + k \ln k \leq n$
- LLL parameter $\delta \in (1/4, 1)$

Output: basis B' of L such that $\|\mathbf{b}'_1\| \leq (k/6)^{(n-1)/2k} \lambda_1(L)$

```
procedure RSR(B, k, δ)
    (B, 𝔟, R) ← LLL(B, δ)                    /* B = B̂R, 𝔟 = (‖b̂_1‖², ..., ‖b̂_n‖²) */
    while ‖b̂_n‖² < (6/k)^{n/k} ‖b_1‖² do
        u ← ⌈ k(k−1)/4 · log_2(1/q_B) ⌉ + 1
        repeat
            v ← Sample(B, R, x) for x ∈_R { 0, ..., 2^u − 1 }
        until ‖v‖² < 0.99‖b_1‖²
        (B, 𝔟, R) ← LLL([v, b_1, ..., b_n], δ)
    end while
end procedure
```

RSR consists of two nested loops: The inner loop samples random lattice vectors $\mathbf{v}$ until $\mathbf{v}$ is strictly shorter than the first base vector $\mathbf{b}_1$. The samples are taken from a search space specified in Sect. 2.3. Each call of the Sample algorithm takes $O(n^2)$ time. Schnorr's analysis shows that the chance to find such a vector within at most 2^u iterations is at least $1/2$ if $q_{\mathsf{B}} \leq (6/k)^{1/k}$. The value q_{B} is a property of the basis B defined by (GSA). Thus the inner loop will run for expected $O(n^2(6/k)^{k/4})$ time.

The outer loop prepends the short vector $\mathbf{v}$ to the basis and LLL reduces that generating system. Such an LLL update can be done in $O(n^3)$ time. LLL never replaces the first vector with a longer one, therefore $\|\mathbf{b}_1\|^2$ is reduced in every outer loop iteration at least by the factor 0.99. Since LLL guarantees $\lambda_1(L)^2 \leq \|\mathbf{b}_1\|^2 \leq (\delta - 1/4)^{-(n-1)} \lambda_1(L)^2$, there can be at most $O(n)$ outer loop iterations.

In summary, RSR computes a basis B of L such that $\|\mathbf{b}_1\| \leq (k/6)^{n/2k} \lambda_1(L)$ in expected $O(n^3(k/6)^{k/4} + n^4)$ time.

2.2. The Geometric Series Assumption

We define the Geometric Series Assumption that constrains the lengths $\|\hat{\mathbf{b}}_j\|$ of the orthogonalized base vectors of an LLL reduced basis and we discuss its motivation.

Recall that LLL enforces

$$\|\hat{\mathbf{b}}_j\|^2 \geq (\delta - 1/4) \|\hat{\mathbf{b}}_{j-1}\|^2$$

for all $1 < j \leq n$ and, in consequence,

$$\|\hat{\mathbf{b}}_j\|^2 \geq (\delta - 1/4)^{j-1} \|\mathbf{b}_1\|^2 .$$

I. e., the lengths of the Gram-Schmidt vectors of an LLL reduced basis are bounded from below by a geometric sequence. The same holds for BKZ reduced bases – every BKZ reduced basis is also LLL reduced, after all. In general, one obtains stricter bounds for BKZ reduced bases, though.

Note, these are lower bounds only. LLL does not enforce that $(\|\hat{\mathbf{b}}_j\|^2)_j$ is indeed a geometric sequence. A simple counterexample is the LLL reduced basis

$$\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & n \end{pmatrix} . \tag{2.1}$$

However, if one LLL or BKZ reduces random lattice bases (for some fuzzy definition of "random") then one observes in practice that the lengths of the Gram-Schmidt vectors $\hat{\mathbf{b}}_j$ of the output basis in fact resemble a geometric sequence $(\|\hat{\mathbf{b}}_j\|^2)_j \approx (q^{j-1}\|\mathbf{b}_1\|^2)_j$ for some $q \in [0,1]$. (See, for instance, the top graph in Diagram (a) of Fig. 3.2 on page 28.) Ajtai – who constructs worst-case instances for Schnorr's $2k$ algorithm that have this property – supports this observation based on heuristic arguments [Ajt03].

The *Geometric Series Assumption* states that this outcome is to be expected from the LLL or BKZ reduction of any lattice basis.

Assumption 1 (Geometric Series Assumption (GSA)). *Let* $\mathsf{B} \in \mathbb{Z}^{d\times n}$ *be an* LLL *reduced lattice basis. Then there is* $q_{\mathsf{B}} \in [0,1]$ *such that* $\|\hat{\mathbf{b}}_j\|^2 = q_{\mathsf{B}}^{j-1}\|\mathbf{b}_1\|^2$ *for all* $j = 1, \dots, n$.

Under (GSA), q_{B} is uniquely determined by the lattice basis B. We therefore call q_{B} the *GSA coefficient* of B.

2.3. The Candidate Set

Here we define the subset of $L(\mathsf{B})$ where RSR samples vectors from and describe the algorithm that is used to generate the samples.

Let $\mathbf{v} = \mathsf{B}\mathbf{x} \in L(\mathsf{B})$ be a point in the lattice generated by the basis $\mathsf{B} \in \mathbb{Z}^{d\times n}$ with Gram-Schmidt decomposition $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$. We can represent $\mathbf{v}$ in terms of the vector space basis $\hat{\mathsf{B}}$ as well, namely $\mathbf{v} = \hat{\mathsf{B}}\boldsymbol{\nu}$ with $\boldsymbol{\nu} = (\nu_1, \dots, \nu_n)^t = \mathsf{R}\mathbf{x} \in \mathbb{Q}^n$. The $\hat{\mathbf{b}}_j$ are pairwise orthogonal whence

$$\|\mathbf{v}\|^2 = \sum_{j=1}^{n} \nu_j^2 \|\hat{\mathbf{b}}_j\|^2 . \tag{2.2}$$

RSR attempts to find short lattice vectors by sampling elements of a candidate set $S \subset L(\mathsf{B})$. It is obvious from equation (2.2) that for a lattice vector to be short the $|\nu_j|$ need to be as small as possible. However, the orthogonalized base vectors

contribute to the overall length to different degrees; the smaller $\|\hat{\mathbf{b}}_j\|$ the more leeway there is for ν_j.

If B is LLL reduced, then under (GSA) the last orthogonalized base vectors are also the shortest ones. It is therefore plausible to assume that a vector $\mathbf{v} = \sum_{j=1}^{n} \nu_j \hat{\mathbf{b}}_j$ such that

$$\nu_j \in \begin{cases} (-1/2, 1/2] & \text{if } 1 \leq j < n-u, \\ (-1, 1] & \text{if } n-u \leq j < n, \\ \{\, 1 \,\} & \text{if } j = n. \end{cases} \tag{2.3}$$

for all $j = 1, \ldots, n$ and some $1 \leq u \leq n$ is likely to be short. Thus, RSR samples lattice vectors from the set

$$S_{u,\mathsf{B}} := \{\, \mathbf{v} \in L(\mathsf{B}) \mid \mathbf{v} = \sum_{j=1}^{n} \nu_j \hat{\mathbf{b}}_j \text{ and all } \nu_j \text{ subject to (2.3)} \,\} \tag{2.4}$$

where the parameter u is fixed in every outer loop iteration of RSR depending on q_{B} and the RSR parameter k. We will determine the actual probability that a vector in $S_{u,\mathsf{B}}$ is sufficiently short in Sect. 2.5.

The Gram-Schmidt coefficient matrix R of B is unit upper triangular. That enables us to efficiently enumerate $S_{u,\mathsf{B}}$ by the algorithm Sample, Alg. 2 on the facing page. Sample is almost the same as Schnorr's algorithm SA except that the random coin toss in the latter is replaced by the binary digits of the integer input argument x.

Proposition 7. *Let* $\mathsf{B} \in \mathbb{Z}^{d \times n}$ *with Gram-Schmidt decomposition* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$ *and* $u \in \{\, 1, \ldots, n \,\}$. *Then* $\mathsf{Sample}(\mathsf{B}, \mathsf{R}, x) \in S_{u,\mathsf{B}}$ *for any* $x \in \{\, 0, \ldots, 2^u - 1 \,\}$.

Proof. In the algorithm, $\mathbf{v}$ is initialized with $\mathbf{b}_n \in L(\mathsf{B})$. It is updated in the loop body only where it is replaced by $\mathbf{v} + y\mathbf{b}_j \in L(\mathsf{B})$ for some integer y and $j \in \{\, 1, \ldots, n-1 \,\}$. Therefore, the final return value $\mathbf{v}$ is indeed a point in $L(\mathsf{B})$.

We state some of the loop invariants and show that they hold before and after each loop iteration.

a) $0 \leq x < 2^{u-n+j+1}$: Before the first iteration we have $j = n-1$ whence $0 \leq x < 2^u = 2^{u-(n-j)+1}$ holds by assumption. In the loop body, x is replaced by $\lfloor x/2 \rfloor$, so from this point on $0 \leq x < 2^{u-n+j}$. None of the variables x, u, n, and j changes until the end of the loop body is reached where j is replaced by $j-1$. Hence, $0 \leq x < 2^{u-n+j+1}$ is part of the loop's postcondition.

b) $\mathbf{v} = \hat{\mathsf{B}}\boldsymbol{\nu}$: The algorithm's precondition $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$ implies $\mathbf{b}_j = \hat{\mathsf{B}}\mathbf{r}_j$ for all $1 \leq j \leq n$. Thus, the invariant is satisfied when the execution reaches the loop since then $\mathbf{v} = \mathbf{b}_n$ and $\boldsymbol{\nu} = \mathbf{r}_n$. Let $\mathbf{v}^{(\text{old})}$ and $\mathbf{r}^{(\text{old})}$ be the values of $\mathbf{v}$ and $\boldsymbol{\nu}$, respectively, when the loop body is entered by. Similarly, let $\mathbf{v}^{(\text{new})}$ and $\mathbf{r}^{(\text{new})}$ be the respective values at the end of the loop body. Then $\mathbf{v}^{(\text{new})} = \mathbf{v}^{(\text{old})} + y\mathbf{b}_j = \hat{\mathsf{B}}\mathbf{r}^{(\text{old})} + y\hat{\mathsf{B}}\mathbf{r}_j = \hat{\mathsf{B}}(\mathbf{r}^{(\text{old})} + y\mathbf{r}_j) = \hat{\mathsf{B}}\mathbf{r}^{(\text{new})}$.

Algorithm 2 Sample

Input:
- unit upper triangular matrix $\mathsf{R} = [\mathbf{r}_1, \ldots, \mathbf{r}_n] \in \mathbb{Q}^{n \times n}$,
- lattice basis $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d \times n}$ with Gram-Schmidt decomposition $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$,
- $x \in \{0, \ldots, 2^{n-1} - 1\}$

Output: $\mathbf{v} \in L(\mathsf{B})$

```
procedure Sample(B, R, x)
    v ← b_n
    ν = (ν_1, ..., ν_n)^t ← r_n                          /* ν_n = 1 */
    for j from n − 1 downto 1 do
        y ← ⌈ν_j − 1/2⌉                                  /* −1/2 < ν_j − y ≤ 1/2 */
        if x = 1 mod 2 then
            if ν_j − y ≤ 0 then
                y ← y − 1                                /* 1/2 < ν_j − y ≤ 1 */
            else
                y ← y + 1                                /* −1 < ν_j − y ≤ −1/2 */
            end if
        end if
        x ← ⌊x/2⌋
        v ← v − y b_j
        ν ← ν − y r_j                                    /* ν_j ← ν_j − y */
    end for
    return v
end procedure
```

c) $\nu_{j+1}, \ldots, \nu_n$ satisfy (2.3): This invariant holds before the first loop iteration because then $j = n - 1$ and $\nu_n = 1$ due to $\boldsymbol{\nu} = \mathbf{r}_n$. The first statement in the loop body sets the integer y such that $\nu_j - y \in (-1/2, 1/2]$. If $j < n - u$, then $0 \leq x < 2^{u-n+j+1} \leq 2^{u-n+(n-u-1)+1} = 2^0$ by invariant (a), i. e., $x = 0$ is even and y won't be changed in this iteration anymore. If $n - u \leq j < n$, then y may be incremented or decremented, but always in such a way that $\nu_j - y \in (-1, 1]$.

The final assignment $\boldsymbol{\nu} \leftarrow \boldsymbol{\nu} - y\mathbf{r}_j$ does not change $\nu_{j+1}, \ldots, \nu_n$ since the kth coefficient of $\mathbf{r}_j$ is 0 for all $j < k \leq n$. ν_j is replaced by $\nu_j - y$ because the jth coefficient of $\mathbf{r}_j$ is 1. Thus, $\nu_j, \nu_{j+1}, \ldots, \nu_n$ satisfy the search space condition (2.3). Even when the loop variable j is decremented at the end of the loop body, the invariant still holds.

When the loop terminates we have $j = 0$, so all $\nu_1, \ldots, \nu_n$ satisfy the search space condition (2.3) by invariant (c). This means the return value of Sample belongs in fact to the search space $S_{u,\mathsf{B}}$ because $\mathbf{v} = \hat{\mathsf{B}}\boldsymbol{\nu} = \sum_{j=1}^{n} \nu_j \hat{\mathbf{b}}_j \in L(\mathsf{B})$ by invariant (b). □

Proposition 8. *Let* $\mathsf{B} \in \mathbb{Z}^{d\times n}$ *with Gram-Schmidt decomposition* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$*. The function*

$$f : \{\, 0, \ldots, 2^u - 1 \,\} \to S_{u,\mathsf{B}} : x \mapsto \mathsf{Sample}(\mathsf{B}, \mathsf{R}, x)$$

is a bijection. In particular, $|S_{u,\mathsf{B}}| = 2^u$*.*
If the matrix R *is given in floating point format, then the computation of the lattice vector* $\mathsf{Sample}(\mathsf{B}, \mathsf{R}, x)$ *requires* $O(n^2)$ *integer and floating point operations for any admissible parameter* x*.*

Proof. Let $x, x' \in \{\, 0, \ldots, 2^u - 1 \,\}$ and $x \neq x'$. Let $k_0 = \max\{\, k \in \mathbb{N} \mid x = x' \bmod 2^k \,\}$. Say $\lfloor x/2^{k_0} \rfloor$ is even and $\lfloor x'/2^{k_0} \rfloor$ is odd. When computing $\mathsf{Sample}(\mathsf{B}, \mathsf{R}, x)$ and $\mathsf{Sample}(\mathsf{B}, \mathsf{R}, x')$, the first k_0 iterations of the loop body will be exactly the same, so $\nu_j = \nu'_j$ for $n - k_0 \leq j \leq n$. But in the subsequent iteration, x will be even and x' will be odd. Thus, $\nu_{n-k_0-1} \in (-1/2, 1/2]$ whereas $\nu'_{n-k_0-1} \in (-1, 1] \setminus (-1/2, 1/2]$. The remaining iterations will not change the ν_j computed so far whence $\mathsf{Sample}(\mathsf{B}, \mathsf{R}, x) = \sum_{j=1}^n \nu_j \hat{\mathbf{b}}_j \neq \sum_{j=1}^n \nu'_j \hat{\mathbf{b}}_j = \mathsf{Sample}(\mathsf{B}, \mathsf{R}, x')$ differ. f is therefore injective.

Let $\widetilde{\mathbf{v}} = \sum_{j=1}^n \widetilde{\nu}_j \hat{\mathbf{b}}_j \in L(\mathsf{B}) \setminus \operatorname{Im} f$. If $\widetilde{\nu}_n \neq 1$, then $\widetilde{\mathbf{v}} \notin S_{u,\mathsf{B}}$. Otherwise, let $k_1 = 1 + \max\{\, k \in \mathbb{N} \mid \exists \mathbf{v} = \sum_{j=1}^n \nu_j \hat{\mathbf{b}}_j \in \operatorname{Im} f : \widetilde{\nu}_j = \nu_j \text{ for } j = n-k, \ldots, n \,\}$, $1 \leq k_1 < n$, and let $\mathbf{v} \in \operatorname{Im} f$ a vector such that the Gram-Schmidt coefficients $\nu_{n-k_1+1}, \ldots, \nu_n$ of $\mathbf{v}$ coincide with the corresponding coefficients of $\widetilde{\mathbf{v}}$. Then $\widetilde{\nu}_{n-k_1} = \nu_{n-k_1} + y$ for some integer $y \neq 0$ because R is unit upper triangular. In particular, $\widetilde{\nu}_{n-k_1} \notin (-1/2, 1/2]$ if $k_1 > u$. Similarly, $\widetilde{\nu}_{n-k_1} \notin (-1, 1]$ if $1 \leq k_1 \leq u$ because else one could construct $\mathbf{v}' = \sum_{j=1}^n \nu'_j \hat{\mathbf{b}}_j \in \operatorname{Im} f$ such that $\widetilde{\nu}_j = \nu'_j$ for $j = n - k_1, \ldots, n$ in violation of the definition of k_1. (One only needs to flip the ${k_1}^{\text{th}}$ bit in the preimage of $\mathbf{v}$ to find the preimage of $\mathbf{v}'$.) Thus, $\widetilde{\nu}_{n-k_1}$ cannot satisfy the search space condition (2.3) and $\widetilde{\mathbf{v}} \notin S_{u,\mathsf{B}}$. f is therefore surjective.

Sample performs $O(n)$ loop iterations and each iteration is dominated by an update of both an integer and a floating point vector of length n. So the total cost of Sample is $O(n^2)$ integer and floating point operations. □

2.4. The Randomness Assumption

The analysis of RSR treats the Gram-Schmidt coefficients of the vectors sampled by Sample as independent uniform random variables. We formally state in this subsection the underlying assumption and discuss its validity.

Assumption 2 (Randomness Assumption (RA)). *Let* $\mathsf{B} \in \mathbb{Z}^{d\times n}$ *with Gram-Schmidt decomposition* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$*. Consider the random functions* ν_j*,* $1 \leq j < n$*, defined by* $\sum_{j=1}^n \nu_j \hat{\mathbf{b}}_j = \mathsf{Sample}(\mathsf{B}, \mathsf{R}, x)$ *for uniform random* $x \in_R \{\, 0, \ldots, 2^u - 1 \,\}$*.*

Then the ν_j *are statistically independent and uniformly distributed random variables in* $(-1/2, 1/2]$ *if* $j < n - u$*, and in* $(-1, 1]$ *if* $n - u \leq j < n$*.*

It is obvious that (RA) cannot strictly hold: Given B, ν_{n-1} can take only two values, so it will not pass any uniformity test in the interval $(-1, 1]$. Similarly, ν_{n-2}

can take at most four values. Once we know ν_{n-1}, there are only two values left for ν_{n-2}, so statistical tests will recognize them as not independent. How can (RA) be justified for the analysis of RSR anyway?

The point is that only the very last ν_j will fail simple statistical tests. If $j < n-u$, then ν_j can take 2^u different values; if only one of the $\nu_{n-u}, \ldots, \nu_{n-1}$ changes, then ν_j will take a different value. Simple tests cannot tell the ν_j apart from independent uniform random variables unless $n-j$ is too small. So (RA) makes sense for ν_j with $j < n-u$.

(RA) is used to estimate the expected length of sampled vectors. The fact that (RA) does not really hold for ν_j with j near n might skew the results to some degree. However, this failure takes place in those base columns that contribute the least to the overall length because of (2.2) and (GSA). Therefore, the effect is negligible provided u is not too small, say $u \geq 15$.

2.5. Analysis of RSR

In this subsection we present the most important parts of Schnorr's analysis of RSR. In particular, we explain why, under (GSA) and (RA), the expected number of inner loop iterations in RSR is $O((k/6)^{k/4})$. We also explain in more detail than above why RSR terminates after $O(n)$ outer loop iterations and why its postcondition is satisfied as soon as the loop is left.

The inner RSR loop. We estimate the probability that a single sample $\mathbf{v}$, computed by $\mathsf{Sample}(\mathsf{B}, \mathsf{R}, x)$ is short enough to terminate the inner RSR loop. This enables us to determine the expected number of iterations of the loop.

Theorem 9 (Schnorr). *Let $k, n \in \mathbb{N}$ such that $24 \leq k$ and $3k + \frac{1}{4}k\log_2 k - 13 \leq n$. Let $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R} \in \mathbb{Z}^{d\times n}$ be a lattice basis subject to (GSA) with GSA coefficient $q_{\mathsf{B}} \leq (6/k)^{1/k}$. Let $\mathbf{v}$ be uniformly sampled from $S_{u,\mathsf{B}}$ where $u = \left\lceil \frac{k(k-1)}{4}\log_2(1/q_{\mathsf{B}})\right\rceil + 1$. Under (RA), we have*

$$\Pr\left[\,\|\mathbf{v}\|^2 \leq 0.971\|\mathbf{b}_1\|^2\,\right] \geq \frac{1}{2}\left(\frac{6}{k}\right)^{\frac{k-1}{4}} \geq \frac{1}{2}q_{\mathsf{B}}^{\frac{k(k-1)}{4}}.$$

The probability, under (RA), that the inner RSR loop terminates after not more than $\lceil 2(k/6)^{(k-1)/4}\rceil \leq 2^u$ iterations is greater than $0.63 > 1/2$.

In fact, our statement of Theorem 9 is slightly stronger than [Sch03, Theorem 1]. As it turns out, we can somewhat relax the upper bound on k (that Schnorr states within his proof) and improve the reduction factor from 0.99 to 0.971. We do not see practical consequences, though: For large k the expected running time of the inner loop becomes soon impractical (e. g., if $k \geq 60$ then $u \geq 50$). And relaxing the reduction factor to 0.99 does not yield a smaller minimal value for u, at least not with this approach.

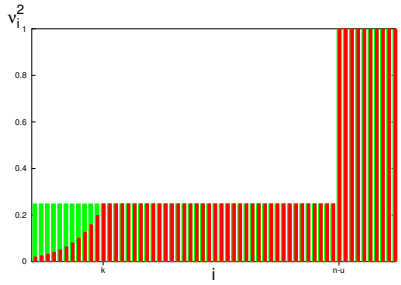


Figure 2.1.: The event $(\mathcal{S}_{k,r})$.
■: possible range of ν_j^2 for arbitrary $\mathbf{v} \in S_{u,\mathsf{B}}$.
■: possible range of ν_j^2 in case of $(\mathcal{S}_{k,r})$.

The proof can be summarized as follows: We define an event $(\mathcal{S}_{k,r})$ and compute in Lemma 12 the conditional mean value $E\left[\,\|\mathbf{v}\|^2 \,\middle|\, (\mathcal{S}_{k,q_\mathsf{B}})\,\right]$. In Lemma 13, the conditional mean value $E\left[\,\|\mathbf{v}\|^2 \,\middle|\, (\mathcal{S}_{k,f(k)})\,\right]$ with $f(k) = (6/k)^{1/k} \geq q_\mathsf{B}$ turns out to be shorter than $0.971\|\mathbf{b}_1\|^2$ under the preconditions of Theorem 9. Therefore, by elementary probability theory, $1/2 \Pr\left[\,(\mathcal{S}_{k,f(k)})\,\right]$ is a lower bound for the success probability $\Pr\left[\,\|\mathbf{v}\|^2 \leq 0.971\|\mathbf{b}_1\|^2\,\right]$ of a single sample. From that we conclude by Lemma 14 the stated lower bound for the number of samples we need in order to push the success probability to 0.63.

Definition 10. *Let* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R} \in \mathbb{Z}^{d\times n}$ *be a lattice basis,* $1 \leq u < n$, $1 < k < n-u$, *and* $r \in [0,1]$. *Let* $\mathbf{v} = \sum_{j=1}^{n} \nu_j \hat{\mathbf{b}}_j = \mathsf{Sample}(\mathsf{B},\mathsf{R},x)$ *for random* $x \in_\mathsf{R} \{\,0,\ldots,2^u-1\,\}$. *The event* $(\mathcal{S}_{k,r})$ *is defined by*

$$\nu_j^2 \leq \tfrac{1}{4} r^{k-j} \qquad \textit{for all } j \in \{\,1,\ldots,k\,\}. \tag{$\mathcal{S}_{k,r}$}$$

Recall that the definition of the search space $S_{u,\mathsf{B}}$ made sure the ν_j^2 are rather small. I. e., we already know by (2.3) that $\nu_j^2 \leq 1/4$ if $1 \leq j \leq k$. If $(\mathcal{S}_{k,r})$ occurs, then the first k Gram-Schmidt coefficients of the sampled vector are particularly small where "particular" is specified by r: The smaller r the smaller are the ν_j^2. $(\mathcal{S}_{k,r})$ is illustrated by Fig. 2.1.

Lemma 10. *Let* X *be a random variable uniformly distributed in* $(-t,t] \subset \mathbb{R}$. *Then the probability function and mean value of* X^2 *are*

$$\Pr\left[\,X^2 \leq r\,\right] = \begin{cases} 0 & \textit{if } r \leq 0, \\ \frac{\sqrt{r}}{t} & \textit{if } 0 < r < t^2, \\ 1 & \textit{if } t^2 \leq r, \end{cases} \quad \textit{and} \quad E\left[\,X^2\,\right] = \tfrac{1}{3}t^2\,, \tag{2.5}$$

respectively.

Proof. Set $I = (-t, t]$ and let λ denominate the Lebesgue measure. The probability measure σ of the uniformly distributed X is $\sigma(J) = \lambda(J \cap I)/\lambda(I)$ for any $J \subset \mathbb{R}$. Thus,

$$\Pr\left[\, X^2 \leq r \,\right] = \sigma([-\sqrt{r}, \sqrt{r}]) = \frac{2\sqrt{r}}{2t} = \frac{\sqrt{r}}{t}$$

for $0 < r < t^2$ and

$$E\left[\, X^2 \,\right] = \int_{\mathbb{R}} X^2 \, d\sigma = \frac{1}{\lambda(I)} \int_I X^2 \, d\lambda = \frac{1}{2t} \left[\tfrac{1}{3} X^3\right]_{-t}^{t} = \tfrac{1}{3} t^2 \,.$$

□

Lemma 11. *Let* B*,* k*,* u*, and* r *as in Def. 10. Then, under (RA),*

$$\Pr\left[\, (\mathcal{S}_{k,r}) \,\right] = r^{\frac{k(k-1)}{4}} \,.$$

Proof. For any uniform random variable $X \in_R (-t, t]$ and bound $R \in [0, t^2]$, the probability of $X^2 \leq R$ is $\sqrt{R/t^2}$ by Lemma 10. For $j \in \{\, 1, \ldots, k \,\}$, the $\nu_j \in_R (-1/2, 1/2]$ are uniformly distributed and independent according to (RA). Hence,

$$\Pr\left[\, (\mathcal{S}_{k,r}) \,\right] = \prod_{j=1}^{k} \Pr\left[\, \nu_j^2 \leq \tfrac{1}{4} r^{k-j} \,\right] = \prod_{j=1}^{k} r^{\frac{k-j}{2}} = r^{\frac{k(k-1)}{4}} \,.$$

□

Lemma 12. *Let* B*,* q_{B}*,* n *as in Theorem 9 and let* $k, u \in \mathbb{N}$ *such that* $1 < k < n - u < n$*. Then, under (RA),*

$$E\left[\, \|\mathbf{v}\|^2 \,\middle|\, (\mathcal{S}_{k,q_{\mathsf{B}}}) \,\right] = \frac{k q_{\mathsf{B}}^{k-1} - (k-1) q_{\mathsf{B}}^{k} + 3 q_{\mathsf{B}}^{n-u-1} + 8 q_{\mathsf{B}}^{n-1} - 12 q_{\mathsf{B}}^{n}}{12(1 - q_{\mathsf{B}})} \|\mathbf{b}_1\|^2 \,.$$

Proof. According to (RA), the ν_j are uniformly distributed. Using (GSA), (2.3), and the definition of $(\mathcal{S}_{k,r})$, Lemma 10 implies

$$E\left[\, \nu_j^2 \frac{\|\hat{\mathbf{b}}_j\|^2}{\|\mathbf{b}_1\|^2} \,\middle|\, (\mathcal{S}_{k,r}) \,\right] = \begin{cases} \frac{1}{12} r^{k-j} q_{\mathsf{B}}^{j-1} & \text{if } 1 \leq j < k, \\ \frac{1}{12} q_{\mathsf{B}}^{j-1} & \text{if } k \leq j < n-u, \\ \frac{1}{3} q_{\mathsf{B}}^{j-1} & \text{if } n-u \leq j < n, \\ q_{\mathsf{B}}^{n-1} & \text{if } j = n. \end{cases} \tag{2.6}$$

In particular, $E\left[\, \nu_j^2 \frac{\|\hat{\mathbf{b}}_j\|^2}{\|\mathbf{b}_1\|^2} \,\middle|\, (\mathcal{S}_{k,q_{\mathsf{B}}}) \,\right] = q_{\mathsf{B}}^{k-1}$ for all $j \in \{\, 1, \ldots, k-1 \,\}$. Again by (RA),

the ν_j are statistically independent. Therefore,

$$\begin{aligned}
E\left[\frac{\|\mathbf{v}\|^2}{\|\mathbf{b}_1\|^2}\,\middle|\,(\mathcal{S}_{k,q_\mathsf{B}})\right] &= \sum_{j=1}^{n} E\left[\nu_j^2\frac{\|\hat{\mathbf{b}}_j\|^2}{\|\mathbf{b}_1\|^2}\,\middle|\,(\mathcal{S}_{k,q_\mathsf{B}})\right] \\
&= \frac{1}{12}\sum_{j=1}^{k-1} q_\mathsf{B}^{k-1} + \frac{1}{12}\sum_{j=k}^{n-u-1} q_\mathsf{B}^{j-1} + \frac{1}{3}\sum_{j=n-u}^{n-1} q_\mathsf{B}^{j-1} + q_\mathsf{B}^{n-1} \\
&= \frac{(k-1)q_\mathsf{B}^{k-1}}{12} + \frac{q_\mathsf{B}^{k-1}-q_\mathsf{B}^{n-u-1}}{12(1-q_\mathsf{B})} + \frac{q_\mathsf{B}^{n-u-1}-q_\mathsf{B}^{n-1}}{3(1-q_\mathsf{B})} + q_\mathsf{B}^{n-1} \\
&= \frac{kq_\mathsf{B}^{k-1}-(k-1)q_\mathsf{B}^{k}+3q_\mathsf{B}^{n-u-1}+8q_\mathsf{B}^{n-1}-12q_\mathsf{B}^{n}}{12(1-q_\mathsf{B})}\,.
\end{aligned}$$

□

Lemma 13. *Let $f:\mathbb{R}_+\to\mathbb{R}: x\mapsto (6/x)^{1/x}$. Under the preconditions of Theorem 9, $E\left[\,\|\mathbf{v}\|^2\,\middle|\,(\mathcal{S}_{k,q_\mathsf{B}})\,\right] \le E\left[\,\|\mathbf{v}\|^2\,\middle|\,(\mathcal{S}_{k,f(k)})\,\right] \le 0.971\|\mathbf{b}_1\|^2$.*

Proof. Derivation shows that f has its only global minimum in $x=6e$ where $f(6e)\approx 0.94$ and f has no local maximum. Since $\lim_{x\to\infty} f(x)=1$, this implies $0.94<f(x)<1$ for all $x\in(6,\infty)$.

First, consider the special case $q_\mathsf{B}=f(k)$. We know $8f(k)^{n-1}-12f(k)^n<0$ because of $2/3<f(k)$. Furthermore, the inequation $n-u-1\ge n-\frac{1}{4}k\log_2 k+13$ holds for all $k\ge 24$. This implies $n-u-1\ge 3k$ under the theorem's preconditions on k. Hence we are left with

$$\begin{aligned}
\frac{E\left[\,\|\mathbf{v}\|^2\,\middle|\,(\mathcal{S}_{k,f(k)})\,\right]}{\|\mathbf{b}_1\|^2} &\le \frac{kf(k)^{k-1}-(k-1)f(k)^k+3f(k)^{n-u-1}}{12(1-f(k))} \\
&\le \frac{kf(k)^{k-1}-(k-1)f(k)^k+3f(k)^{3k}}{12(1-f(k))} \qquad (2.7)\\
&= \frac{1}{2f(k)} + \frac{1}{2k(1-f(k))} + \frac{54}{k^3(1-f(k))}\,.
\end{aligned}$$

We already know that $f(k)$ is strictly increasing for $k\ge 24$. Again by derivation, we see that the same holds for $k(1-f(k))$ and $k^3(1-f(k))$. Therefore, the most right hand side in (2.7) is strictly decreasing for $k\ge 24$. Evaluating in $k=24$ yields

$$E\left[\,\|\mathbf{v}\|^2\,\middle|\,(\mathcal{S}_{k,f(k)})\,\right] \le 0.971\|\mathbf{b}_1\|^2\,.$$

(Evaluating in $k=23$ already yields a value $>0.99\|\mathbf{b}_1\|^2$.)

Now consider all admissible q_B. Because of (RA), we have the conditional mean value $E\left[\,\|\mathbf{v}\|^2\,\middle|\,(\mathcal{S}_{k,r})\,\right]=\sum_{j=1}^{n} E\left[\nu_j^2\|\hat{\mathbf{b}}_j\|^2\,\middle|\,(\mathcal{S}_{k,r})\right]$. According to (2.6), each summand $E\left[\nu_j^2\|\hat{\mathbf{b}}_j\|^2\,\middle|\,(\mathcal{S}_{k,r})\right]$ is strictly increasing in both r and q_B. Therefore,

$$E\left[\,\|\mathbf{v}\|^2\,\middle|\,(\mathcal{S}_{k,q_\mathsf{B}})\,\right] \le E\left[\,\|\mathbf{v}\|^2\,\middle|\,(\mathcal{S}_{k,f(k)})\,\right] \le 0.971\|\mathbf{b}_1\|^2$$

for all $q_\mathsf{B}\le f(k)=(6/k)^{1/k}$, $k\ge 24$. □

Lemma 14 (Probability Enhancement). *Let M be a set, $\mathcal{P} : M \to \{0,1\}$ a predicate on M, and X a random variable on M such that $\Pr[\mathcal{P}(X) = 1] > 0$. Let $X_1, \ldots, X_m$ be independent samples of X.*
If $m \geq 1/\Pr[\mathcal{P}(X) = 1]$, then

$$\Pr[\exists j \in \{1, \ldots, m\} : \mathcal{P}(X_j) = 1] > 0.63 > 1/2\,.$$

Proof. The function $g : (1, \infty) \to \mathbb{R} : x \mapsto (1 - 1/x)^x$ is strictly increasing and goes to $\lim_{x\to\infty} g(x) = e^{-1}$ (e. g., [FK90, §2.2]). Therefore, $(1 - 1/x)^m < e^{-1}$ for all $0 < x \leq m$. In particular,

$$\Pr[\exists j \in \{1, \ldots, m\} : \mathcal{P}(X_j) = 1] = 1 - (1 - \Pr[\mathcal{P}(X) = 1])^m > 1 - e^{-1} > 0.63$$

provided $m \geq 1/\Pr[\mathcal{P}(X) = 1]$. □

Now the proof of Schnorr's theorem regarding the success probability of the sampling loop is merely an application of Lemmata 11, 13, and 14.

Proof of Theorem 9. Let $f(k) = (6/k)^{1/k}$. If a sampled vector $\mathbf{v}$ satisfies $(\mathcal{S}_{k,f(k)})$, then we expect (i. e., with probability $> 1/2$) that $\|\mathbf{v}\|^2 \leq 0.971\|\mathbf{b}_1\|^2$ by Lemma 13. The total probability theorem implies

$$\Pr\left[\,\|\mathbf{v}\|^2 \leq 0.971\|\mathbf{b}_1\|^2\,\right] \geq \underbrace{\Pr\left[\,\|\mathbf{v}\|^2 \leq 0.971\|\mathbf{b}_1\|^2 \,\middle|\, (\mathcal{S}_{k,f(k)})\,\right]}_{\geq 1/2} \cdot \underbrace{\Pr\left[\,(\mathcal{S}_{k,f(k)})\,\right]}_{f(k)^{\frac{k(k-1)}{4}}} \tag{2.8}$$

where the event probability $f(k)^{k(k-1)/4} = (6/k)^{(k-1)/4}$ is given by Lemma 11. Therefore, we need at most $\lceil 2(k/6)^{(k-1)/4} \rceil \leq 2^u$ iterations of the inner RSR loop in order to enhance the success probability to more than 0.63 (Lemma 14). □

The outer RSR loop. Theorem 9 bounds the expected running time of the inner RSR loop. The analysis of RSR is completed by a bound on the number of iterations of the outer loop and a check that the algorithm meets indeed its stated postcondition when the outer loop is left.

Lemma 15. *Let k, δ, B as in the preconditions of Alg. 1. The number of iterations of the outer loop of RSR is $O(n)$. When the outer loop is left, then $\|\mathbf{b}_1\| \leq (k/6)^{(n-1)/2k}\lambda_1(L)$.*

Proof. When the outer loop is entered for the first time, then B is already LLL reduced and $\lambda_1(L)^2 \leq \|\mathbf{b}_1\|^2 \leq (\delta - 1/4)^{-(n-1)}\lambda_1(L)^2$. Each iteration prepends a vector $\mathbf{v}$ to the basis such that $\|\mathbf{v}\|^2 \leq 0.99\|\mathbf{b}_1\|^2$. The subsequent LLL reduction never replaces the first vector of the generating system by a longer one. Therefore, after the jth iteration, $\lambda_1(L)^2 \leq \|\mathbf{b}_1\|^2 \leq 0.99^j(\delta - 1/4)^{-(n-1)}\lambda_1(L)^2$. By the very definition of $\lambda_1(L)$, the algorithm cannot produce a lattice vector shorter than $\lambda_1(L)$, so necessarily $0.99^j \geq (\delta - 1/4)^{n-1}$, i. e., $j \in O(n)$.

For any basis $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$ of the lattice L, the lengths of the orthogonalized base vectors provide a lower bound $\lambda_1(L) \geq \min_j \|\hat{\mathbf{b}}_j\|$. (GSA) implies $\min_{j=1,\ldots,n} \|\hat{\mathbf{b}}_j\| = \|\hat{\mathbf{b}}_n\| = q_{\mathsf{B}}^{(n-1)/2}\|\mathbf{b}_1\|$. Therefore, $\|\mathbf{b}_1\| \leq q_{\mathsf{B}}^{-(n-1)/2}\lambda_1(L)$.

The outer loop is left if and only if $\|\hat{\mathbf{b}}_n\|^2 \geq (6/k)^{(n-1)/k}\|\mathbf{b}_1\|^2$, i. e., if $q_{\mathsf{B}} \geq (6/k)^{1/k}$. Then $\|\mathbf{b}_1\| \leq q_{\mathsf{B}}^{-(n-1)/2}\lambda_1(L) \leq (k/6)^{(n-1)/2k}\lambda_1(L)$. □

2.6. Problems with RSR in Practice

Schnorr's RSR makes two assumptions that both do not hold in general. We discuss below how this affects the practicability of RSR. We prove Lemma 16 that yields a sufficient condition on the basis B and the parameter q such that $\frac{1}{2}q^{k(k-1)/4}$ is indeed a lower bound on the success probability of the inner sampling loop. Lemma 16 makes it possible to implement RSR since we now are able to choose a suitable value for q if the basis B does not strictly satisfy (GSA). We exemplify that – even with Lemma 16 – RSR requires soon impractically large parameters and needs to terminate early even though the sampling loop is in fact still likely to find vectors shorter than $\mathbf{b}_1$.

The Randomness Assumption. The RSR algorithm does not explicitly refer to (RA). The assumption is nevertheless crucial for the total correctness of RSR: By Theorem 9, the probability is at least 1/2 that the inner loop will terminate within $\lceil 2q^{-k(k-1)/4} \rceil \leq 2^u$ iterations. Under (RA), if the loop does not terminate after that many iterations, we simply keep sampling lattice points and will eventually find a sufficiently short vector, due to the law of large numbers.

However, our search space is actually finite. We do not even know for sure that it contains such a vector we are after. If it does not, then the inner RSR loop is infinite – RSR as described in [Sch03] can only be partially correct in practice. There are two approaches to overcome this problem:

a) Abort the algorithm if the search space $S_{u,\mathsf{B}}$ is exhausted. This is the most simple solution that makes the algorithm terminate for any admissible input but it has a serious drawback: The probability that RSR will terminate regularly and therefore meet its postcondition is only $2^{-O(n)}$ if the inner loop aborts after $O(2^u)$ iterations.

b) Extend the search space, e. g., search $S_{u+j,\mathsf{B}}$, $j = 1, 2, \ldots$, once $S_{u,\mathsf{B}}$ is exhausted. The proof of Theorem 9 works for larger values of u as well. In fact, it is only the proof of Lemma 13 that imposes the upper bound $u \leq n - 3k - 1$. Each time we extend the search space there will again be a more than 63 % chance that the loop will find a sufficiently short vector within $2q_{\mathsf{B}}^{-k(k-1)/4}$ iterations.

If we implement the latter approach, then, in theory, the algorithm will eventually proceed. In practice, a user can or wants to afford only so many iterations

whence there has to be a limit after which the algorithms gives up. A reasonable implementation will therefore combine both approaches.

The Geometric Series Assumption. It is obvious that the lattice bases encountered in practice do not satisfy (GSA) in the strict sense. The best we can hope for is that the lengths of their orthogonalized vectors $\hat{\mathbf{b}}_j$ *approximate* a geometric sequence. This raises two questions: How much may the bases deviate from (GSA) without rendering the analysis in Sect. 2.5 invalid? And how do we compute the value q_{B} required in the RSR algorithm?

The Lemma 16 below provides a sufficient condition that ensures the inner RSR loop will succeed under (RA) with probability greater than $1/2$ even if B does not satisfy (GSA). We can also infer from this condition how to compute a suitable value for q_{B}. However, without (GSA) we need to modify the loop condition of the outer RSR loop and we can no longer guarantee that $\|\mathbf{b}_1\|^2 \leq (k/6)^{(n-1)/2k}\lambda_1(L)$ once the algorithm terminates.

Under (GSA), Schnorr's event $(\mathcal{S}_{k,r})$ restrict those components of the sampled vector $\mathbf{v}$ that contribute most to its length. Without (GSA), we have no a priori knowledge about the relative lengths of the $\hat{\mathbf{b}}_j$. On the other hand, the expected length of a sampled vector is invariant under any permutation of the first $n-u-1$ Gram-Schmidt vectors. We therefore consider a modified event that takes the lengths of the Gram-Schmidt vectors into account.

Definition 11. *Let* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R} \in \mathbb{Z}^{d\times n}$ *be a lattice basis,* $1 \leq u < n$, $1 < k < n-u$, *and* $r \in [0,1]$. *Let* $\mathbf{v} = \sum_{j=1}^{n} \nu_j \hat{\mathbf{b}}_j = \mathsf{Sample}(\mathsf{B}, \mathsf{R}, x)$ *for random* $x \in_R \{0, \ldots, 2^u - 1\}$. *The permutation* $\sigma_{u,\mathsf{B}} \in \mathrm{Sym}(n)$ *sorts the first* $n-u-1$ *Gram-Schmidt vectors by non-increasing length, i. e.,*

$$\begin{aligned} \|\hat{\mathbf{b}}_{\sigma_{u,\mathsf{B}}(j)}\|^2 \geq \|\hat{\mathbf{b}}_{\sigma_{u,\mathsf{B}}(j+1)}\|^2 \quad & \textit{for } j \in \{1, \ldots, n-u-2\}, \\ \sigma_{u,\mathsf{B}}(j) = j \quad & \textit{for } j \in \{n-u, \ldots, n\}. \end{aligned} \tag{2.9}$$

The event $(\mathcal{E}_{k,u,r})$ *is defined by*

$$\nu^2_{\sigma_{u,\mathsf{B}}(j)} \leq \tfrac{1}{4} r^{k-j} \qquad \textit{for all } j \in \{1, \ldots, k\}. \tag{$\mathcal{E}_{k,u,r}$}$$

Remark 1. If B is subject to (GSA), then, for any u, $\sigma_{u,\mathsf{B}}$ is the trivial permutation $\mathrm{id} : j \mapsto j$ and $(\mathcal{E}_{k,u,r})$ coincides with $(\mathcal{S}_{k,r})$.

Remark 2. All ν_j, $j \in \{1, \ldots, n-u-1\}$ are independent uniform random variables on $(-\frac{1}{2}, \frac{1}{2}]$ by (RA) and equation (2.3). Therefore, the proof of Lemma 11 still applies and the event probability

$$\Pr[(\mathcal{E}_{k,u,r})] = r^{\frac{k(k-1)}{4}} \tag{2.10}$$

does not depend on $\sigma_{u,\mathsf{B}}$.

Definition 12. *Let* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R} \in \mathbb{Z}^{d\times n}$ *be a lattice basis not necessarily subject to (GSA),* $u \in \{1,\dots,n\}$*,* $q \in (0,1)$*, and* $\sigma_{u,\mathsf{B}}$ *as in Def. 11. Using the shorthand* $\delta_j = \delta_j(\mathsf{B},q) = \|\hat{\mathbf{b}}_{\sigma_{u,\mathsf{B}}(j)}\|^2/\|\mathbf{b}_1\|^2 - q^{j-1}$*,* $j = 1,\dots,n$*, and* $k \in \{1,\dots,n-u-1\}$ *define the* deviation $\Delta_{\mathsf{B}}(k,u,q)$ *of* B *as*

$$\Delta_{\mathsf{B}}(k,u,q) := \frac{1}{12}\sum_{j=1}^{k-1} q^{k-j}\delta_j + \frac{1}{12}\sum_{j=k}^{n-u-1}\delta_j + \frac{1}{3}\sum_{j=n-u}^{n-1}\delta_j + \delta_n\,. \tag{2.11}$$

The deviation Δ_{B} is not a proper measure how well B approximates a particular geometric sequence. Since positive and negative δ_j compensate, $\Delta_{\mathsf{B}}(k,u,q)$ can be 0 even if $(\|\hat{\mathbf{b}}_{\sigma_{u,\mathsf{B}}(j)}\|^2/\|\mathbf{b}_1\|^2)_j$ does not resemble a geometric sequence at all. In particular, the deviation does not relate to, e. g., the much more intuitive L_q distance, $q \in [1,\infty]$, between the sequences $(\|\hat{\mathbf{b}}_{\sigma_{u,\mathsf{B}}(j)}\|^2/\|\mathbf{b}_1\|^2)_j$ and $(q_{\mathsf{B}}^{j-1})_j$.

Lemma 16. *Let* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R} \in \mathbb{Z}^{d\times n}$ *be a lattice basis not necessarily subject to (GSA). Let* $k \in \mathbb{Z}$ *such that* $24 \le k$ *and* $3k + \frac{1}{4}k\log_2 k - 13 \le n$*. For* $q \in (0,1)$ *set* $u = u(k,q) = \left\lceil \frac{k(k-1)}{4}\log_2(1/q)\right\rceil + 1$*.*

If there is $q_{\mathsf{B}} \in [0,(6/k)^{1/k}]$ *such that the deviation* $\Delta_{\mathsf{B}}(k,u,q_{\mathsf{B}}) \le 0.019$*, then, under (RA),*

$$\Pr\left[\,\|\mathbf{v}\|^2 \le 0.99\|\mathbf{b}_1\|^2\,\right] \ge \tfrac{1}{2}q_{\mathsf{B}}^{\frac{k(k-1)}{4}}$$

for uniformly sampled $\mathbf{v} \in_R S_{u,\mathsf{B}}$*.*

Proof. We have by (2.3), $(\mathcal{E}_{k,u,q_{\mathsf{B}}})$, and Lemma 10

$$E\left[\nu^2_{\sigma_{u,\mathsf{B}}(j)}\frac{\|\hat{\mathbf{b}}_{\sigma_{u,\mathsf{B}}(j)}\|^2}{\|\mathbf{b}_1\|^2}\,\middle|\,(\mathcal{E}_{k,u,q_{\mathsf{B}}})\right] = \begin{cases} \frac{1}{12}q_{\mathsf{B}}^{k-j}(q_{\mathsf{B}}^{j-1} + \delta_{\sigma_{u,\mathsf{B}}(j)}) & \text{if } 1 \le j < k,\\ \frac{1}{12}(q_{\mathsf{B}}^{j-1} + \delta_{\sigma_{u,\mathsf{B}}(j)}) & \text{if } k \le j < n-u,\\ \frac{1}{3}(q_{\mathsf{B}}^{j-1} + \delta_{\sigma_{u,\mathsf{B}}(j)}) & \text{if } n-u \le j < n,\\ q_{\mathsf{B}}^{n-1} + \delta_{\sigma_{u,\mathsf{B}}(n)} & \text{if } j = n. \end{cases}$$

Set

$$s(k,q_{\mathsf{B}},n) := \frac{kq_{\mathsf{B}}^{k-1} - (k-1)q_{\mathsf{B}}^{k} + 3q_{\mathsf{B}}^{n-u-1} + 8q_{\mathsf{B}}^{n-1} - 12q_{\mathsf{B}}^{n}}{12(1-q_{\mathsf{B}})}\,.$$

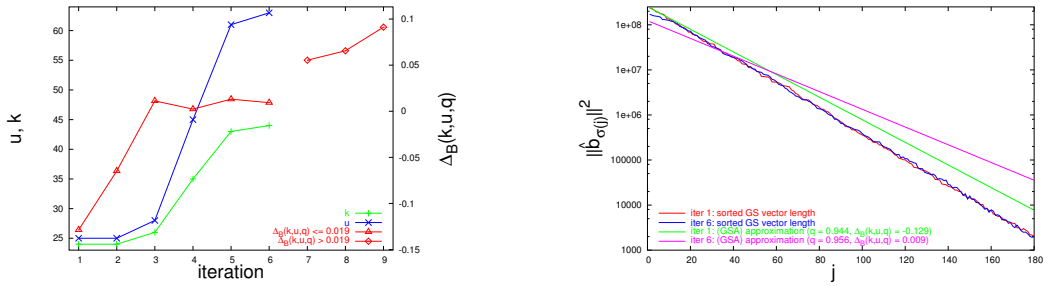
Following the proofs of Lemmata 12 and 13, we find

$$\begin{aligned} E\left[\,\|\mathbf{v}\|^2 \,\middle|\, (\mathcal{E}_{k,u,q_{\mathsf{B}}})\,\right] &= (s(k,q_{\mathsf{B}},n) + \Delta_{\mathsf{B}}(k,u,q_{\mathsf{B}}))\,\|\mathbf{b}_1\|^2\\ &\le (0.971 + \Delta_{\mathsf{B}}(k,q_{\mathsf{B}}))\|\mathbf{b}_1\|^2\\ &\le 0.99\|\mathbf{b}_1\|^2\,. \end{aligned}$$

We can therefore estimate

$$\Pr\left[\,\|\mathbf{v}\|^2 \le 0.99\|\mathbf{b}_1\|^2\,\right] \ge \Pr\left[\,\|\mathbf{v}\|^2 \le 0.99\|\mathbf{b}_1\|^2 \,\middle|\, (\mathcal{E}_{k,u,q_{\mathsf{B}}})\,\right]\cdot\Pr\left[\,(\mathcal{E}_{k,u,q_{\mathsf{B}}})\,\right] \ge \tfrac{1}{2}q_{\mathsf{B}}^{\frac{k(k-1)}{4}}$$

by the total probability theorem. □



(a) Minimum RSR parameters.

(b) (GSA) approximation by minimum RSR parameters.

Figure 2.2.: Typical RSR parameters.

Remark 3. Schnorr asserts in a short remark that his theorems "hold for approximations where $\sum_{i=1}^{n} \mu_i^2(\|\hat{\mathbf{b}}_i\|^2/\|\mathbf{b}_1\|^2 - q^{i-1})$ is sufficiently small for random $\mu_i \in_R [-1/2, 1/2]$, e. g. smaller than 0.01" [Sch03, p. 5]. Even though his statement covers the gist of Lemma 16, it is somewhat imprecise. Schnorr ignores that the effect of any overlength of $\hat{\mathbf{b}}_j$ on the analysis is different depending on j. Furthermore, basing the condition on "random" Gram-Schmidt coefficients is somewhat fuzzy. In contrast, Lemma 16 gives a formula that is based on the – under (RA)– well defined expected values $E\left[\nu_j^2\|\hat{\mathbf{b}}_j\|^2/\|\mathbf{b}_1\|^2 \,\middle|\, (\mathcal{E}_{k,u,q_\mathsf{B}})\right]$, that can easily be evaluated, and that additionally gives better results since $(\mathcal{E}_{k,u,q_\mathsf{B}})$ restricts always those components that contribute most to length of sampled vectors.

Given the RSR parameter k and the lattice basis B, let $q_{\text{opt}} \in (0, (6/k)^{1/k}]$ be the maximum value such that

$$\Delta_\mathsf{B}^{(\mathsf{RSR})}(k, q_{\text{opt}}) = \min\{\, \Delta_\mathsf{B}^{(\mathsf{RSR})}(k, q) \mid q \in (0, (6/k)^{1/k}] \,\}$$

where $\Delta_\mathsf{B}^{(\mathsf{RSR})}(k, q) := \Delta_\mathsf{B}(k, u(k, q), q)$. If we were to implement RSR, then we needed to iterate the outer RSR loop until $\Delta_\mathsf{B}^{(\mathsf{RSR})}(k, q_{\text{opt}}) > 0.019$.

Note that $q \mapsto \Delta_\mathsf{B}^{(\mathsf{RSR})}(k, q)$ is piecewise defined by high degree polynomial functions. Therefore, we do not a priori know that we can find a q_{opt} by numerical methods. The Simple Sampling Reduction algorithm (SSR) that we will propose in Sect. 3.1 will avoid that problem.

Ignoring the fact that in general there might arise numerical problems, we studied which choice of k yields q_{opt} and $\Delta_\mathsf{B}^{(\mathsf{RSR})}(k, q_{\text{opt}})$ subject to the preconditions of Lemma 16. We considered the bases that were generated in the course of Simple Sampling Reduction. SSR is similar enough to RSR that this sequence of bases could also have been generated by RSR. We computed $q_{\text{opt}}(k, \mathsf{B})$ by means of the function `scipy.optimize.brute` from the Python SciPy library [Sci04].)

Fig. 2.2 shows the results we obtained for an example in dimension $n = 180$ that we will further discuss in Sect. 3.1.2. In our experience, this data is representative. In diagram (a), $k_{\min}$ – marked by "+" – is the minimum admissible RSR parameter such that the deviation is sufficiently small, i. e.

$$k_{\min} = \min\{\, k \in \{\, 24, \ldots, n \,\} \mid 3k + \tfrac{1}{4} k \log_2 k - 13 \leq n \text{ and } \Delta_{\mathsf{B}}^{(\mathsf{RSR})}(k, q_{\text{opt}}) \leq 0.019 \,\}\,.$$

The corresponding values for the estimated search space size $u = u(k_{\min}, q_{\text{opt}})$ and the deviation $\Delta_{\mathsf{B}}^{(\mathsf{RSR})}(k_{\min}, q_{\text{opt}})$ are marked by "×" and "△", respectively. In the first three iterations, $k_{\min}$ is not larger than 26. Here $q_{\text{opt}}(k_{\min}, \mathsf{B}) \approx 0.945$ yields deviations not larger than 0.011 and small search space sizes $u \leq 28$. But $k_{\min}$ grows to 35 in the fourth iteration and and to values ≥ 43 after that, $q_{\text{opt}}(k_{\min}, \mathsf{B}) \approx 0.956 \approx (6/k)^{1/k}$. This implies estimated search space sizes $u \geq 63$ which are too large to be enumerated in practice.

In the seventh iteration and afterwards the minimum deviation was too large for all admissible k. Its minimum value, marked by "◊" in diagram (a), rises from 0.055 to 0.163 in the tenth iteration. At this point, RSR could not proceed whereas SSR continued to improve $\mathbf{b}_1$.

We have seen in Lemma 16 that we can loosen the (GSA) for the analysis of the inner loop. However, (GSA) was also instrumental when we computed the approximation factor achieved by RSR. Without (GSA), all we know when the outer loop terminates is – as always – $\min_j \|\hat{\mathbf{b}}_j\|^2 \leq \lambda_1(L)^2$. We no longer have a relation between $\|\mathbf{b}_1\|^2$ and $\min_j \|\hat{\mathbf{b}}_j\|^2$ that is stricter than what the initial LLL reduction implies whence we cannot be sure anymore that $\|\mathbf{b}_1\|^2 \leq (k/6)^{(n-1)/2k} \lambda_1(L)$.

Finally, diagram (b) in Fig. 2.2 exhibits that – in practice – we cannot base our reasoning about the sampling reduction's postcondition w. r. t. the SVP approximation factor on (GSA). The diagram shows the squared lengths of the Gram-Schmidt vectors in iterations 1 and 6, sorted according to $\sigma_{u,\mathsf{B}}$. Superimposed are the hypothetical squared lengths $\|\mathbf{b}_1\|^2 \cdot q_{\text{opt}}(k_{\min}, \mathsf{B})^{j-1}$ that the bases are supposed to approximate according to (GSA). It is apparent that, in the sixth iteration, $\min_j \|\hat{\mathbf{b}}_j\|^2$ is more than one order of magnitude smaller than estimated by the (GSA) approximation. According to the (GSA) approximation, we achieved an SVP approximation factor about $1{\times}10^8/3{\times}10^4 \approx 3{\times}10^3$ whereas the actual data supports only an SVP approximation factor $1{\times}10^8/2.5{\times}10^3 \approx 4{\times}10^4$.

Thus, we cannot assume that RSR in practice meets the bounds given by the theoretical analysis under (GSA) and (RA).

Chapter 3.

Practical Sampling Reduction

We propose a practical Sampling Reduction algorithm and several generalizations and describe their empiric behaviors.

3.1. Simple Sampling Reduction

We present the Simple Sampling Reduction algorithm in Sect. 3.1.1. This modification of RSR does not refer to the (GSA) coefficient q_{B} and is therefore better suited to handle bases that do not closely approximate (GSA).

Finally, we give data on an example that exhibits the typical behavior of SSR. It also demonstrates that we soon encounter bases for which we cannot find a suitable RSR parameter k and a (GSA) coefficient q_{B} such that we could proceed with RSR. SSR does proceed, though.

3.1.1. The Simple Sampling Reduction Algorithm

As pointed out in Sect. 2.6, RSR needs to be modified in practice in order to deal with the finite search space and the (GSA) failure of the lattice bases encountered in practice. This section describes the Simple Sampling Reduction algorithm (SSR) that incorporates such changes. SSR uses an subalgorithm Check Search Space Size (CSSS) that determines whether the success probability of the inner loop is sufficient to justify a further iteration of the outer loop. Here we use CSSS as a black box; we discuss strategies for its implementation in the following sections.

The overall structure of the Simple Sample Reduction (Alg. 3 on the next page) is the same as the structure of RSR: After an initial LLL reduction of the input basis B the algorithm enters an outer loop. Therein it samples vectors $\mathbf{v} \in S_{u,\mathsf{B}}$ for some u until $\mathbf{v}$ is significantly shorter than $\mathbf{b}_1$. Then the generating system formed by prepending the short vector $\mathbf{v}$ to B is LLL reduced again, replacing the basis B.

Looking at the details, however, we see some notable changes which we discuss below.

Input parameter $u_{\max}$. RSR determines the search space size 2^u based on k and the GSA coefficient q_{B}. As we explained in Sect. 2.6, the probability that $S_{u,\mathsf{B}}$ does not contain any sufficiently short vector is non-negligible. Therefore, in practice we

Algorithm 3 SSR

Input:
- basis $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d \times n}$ of lattice L
- search space bound $u_{\max} \in \mathbb{N}$
- LLL parameter $\delta \in (1/4, 1)$

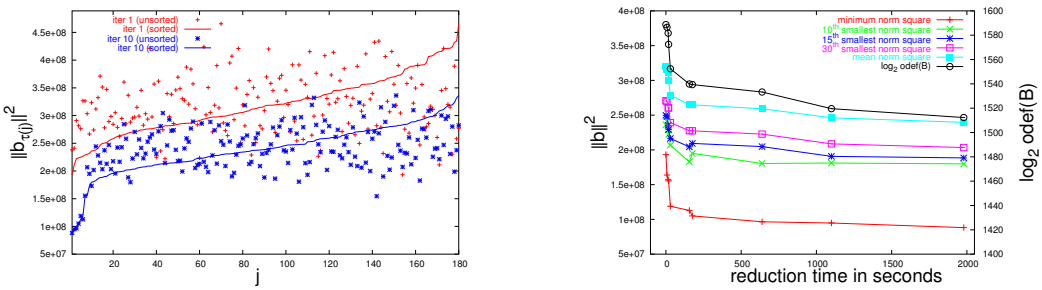
Output: LLL reduced basis $\mathsf{B}' = \hat{\mathsf{B}}'\mathsf{R}'$ of L such that
- $0.99\|\mathbf{b}'_1\|^2 < \min\{\, \|\mathbf{v}\|^2 \mid \mathbf{v} \in S_{u_{\max},\mathsf{B}'} \,\}$ or
- $\mathsf{CSSS}((\|\hat{\mathbf{b}}'_1\|^2, \ldots, \|\hat{\mathbf{b}}'_n\|^2), u_{\max}) = \text{false}$.

```
procedure SSR(B, u_max, δ)
    (B, 𝔟, R) ← LLL(B, δ)                    /* B = B̂R, 𝔟 = (‖b̂_1‖², ..., ‖b̂_n‖²) */
    while CSSS(𝔟, u_max) = true do
        for x from 0 to 2^(u_max) − 1 do
            v ← Sample(B, R, x)
            if ‖v‖² ≤ 0.99‖b_1‖² then
                break
            else if x = 2^(u_max) − 1 then
                terminate("no short vector")
            end if
        end for
        (B, 𝔟, R) ← LLL([v, b_1, ..., b_n], δ)
    end while
    terminate("further progress too unlikely")
end procedure
```

may need to expand the search space and sample more vectors than expected. However, the number of samples a user can or wants to afford is limited. The user specifies this limit by means of the input parameter $u_{\max}$: SSR gives up and terminates after an exhaustive search of $S_{u_{\max},\mathsf{B}}$, i. e., we cap the number of samples at $2^{u_{\max}}$.

Search space enumeration. The vectors from $S_{u,\mathsf{B}}$ are randomly and independently sampled in RSR. This is reasonable because the analysis of the inner RSR loop requires random samples. But it also means that after 2^u samples on average less then $2/3$ of the vectors in $S_{u,\mathsf{B}}$ were actually considered whereas many vectors were sampled twice.

Therefore, SSR deterministically enumerates $S_{u,\mathsf{B}}$. Note that $\mathsf{Sample}(\mathsf{B}, \mathsf{R}, x) \in S_{u,\mathsf{B}}$ for all $u > \log_2 x$ by Proposition 8. That means SSR searches first $S_{u,\mathsf{B}}$ for small u, then for $u+1$, $u+2$, and so on until $u = u_{\max}$. If $u' = \min\{\, u \in \mathbb{N} \mid \exists \mathbf{v} \in S_{u',\mathsf{B}} : \|\mathbf{v}\|^2 \leq 0.99\|\mathbf{b}_1\|^2 \,\} \leq u_{\max}$, then SSR will break out of the inner loop after at most $2^{u'}$ samples. In particular, if random sampling finds a short vector then so will enumeration, and the number of samples required by enumeration is, on average, even smaller by a factor $\approx 2/3$.



(a) Absolute vector lengths before and after 10 SSR iterations.

(b) runtime behavior of SSR.

Figure 3.1.: Typical base vector length reduction by SSR.

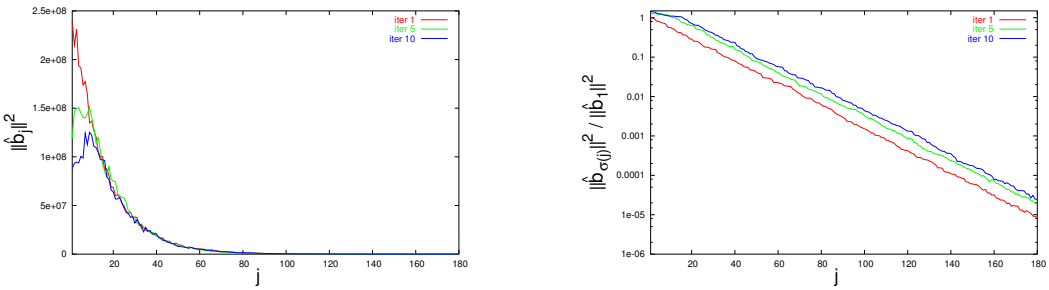
No explicit RSR parameter k. In RSR, k is used (a) for computing the required search space size u and (b) in the loop condition of the outer RSR loop, thereby controlling the approximation factor achieved by RSR. There is no need to compute u in SSR since the maximum search space size is passed in by the user via $u_{\max}$. As we will see later, none of the proposed implementations of CSSS make use of the user defined parameter k. Since we cannot a priori estimate the final approximation factor without (GSA), k has no impact on the postcondition of SSR. Thus, SSR has no use for the parameter k any more.

3.1.2. Practical Behavior of SSR

We will study the empirical behavior of SSR and its variant in more detail later on. Here we only point out on an example what is typical for SSR.

Fig. 3.1 shows the results of SSR applied to a lattice basis in dimension $n = 180$ that were obtained as in Micciancio's variant of the GGH cryptosystem [GGH97, Mic01b]. The original lattice basis O was uniformly chosen from $\{-180, \ldots, 180\}^{180\times 180}$. This basis was transformed into a basis $\mathsf{H} = \mathrm{HNF}(\mathsf{O})$ in Hermite normal form. H was first LLL reduced (initially using software floating point arithmetic) and finally BKZ reduced with $(\delta, \beta) = (0.99, 5)$ using hardware floating point arithmetic. The squared lengths of the columns of the resulting basis B are shown in the upper graph of Fig. 3.1(a). The points in the diagram depict the square lengths in the unpermuted order of the base vectors, i. e., a point with abscissa j represents $\|\hat{\mathbf{b}}_j\|^2$ and $\tau = \mathrm{id}$. The line shows the same data but in non-decreasing order, i. e., here τ is a permutation such that $\|\hat{\mathbf{b}}_{\tau(j)}\|^2 \leq \|\hat{\mathbf{b}}_{\tau(j+1)}\|^2$ for all $j \in \{1, \ldots, n-1\}$.

The lower graph in Fig. 3.1(a) shows the squared lengths of the vectors of the SSR output basis B′ after 10 outer loop iterations. The squared length of the shortest vector in B′ is reduced by a factor $\approx 1/2$ compared to the shortest vector in B. The remaining base vectors also became shorter but less so: the median shrank from



(a) Gram-Schmidt vector lengths.

(b) Relative length of Gram-Schmidt vectors, sorted in non-increasing order.

Figure 3.2.: Typical (GSA) behavior of SSR bases.

3.12×10^8 to 2.36×10^8, i. e., by a factor $\approx 3/4$. Only the first ten base vectors are particularly small. It stands out that the variance of the squared lengths became smaller, in particular if we ignore the first ten base vectors.

We see in Fig. 3.1(b) that most of the reduction happened in the first six iterations within 48 seconds. The following iterations improved the minimum norm square by less than 10 % each and required several minutes of sampling (at a rate of $\approx$ 5200 samples per second on a 2.4 GHz Intel Pentium 4 machine with 1 GByte RAM).

In Fig. 3.2 we see the (GSA) behavior of the lattice basis in the course of SSR. Diagram (a) shows that the input basis B approximates (GSA) well. However, inserting the short vectors found by sampling and re-reducing the generating system with BKZ, $(\delta, \beta) = (0.99, 5)$ makes only the very first Gram-Schmidt vectors shorter. In the end, the graph of $\|\hat{\mathbf{b}}_j\|^2$ forms a distinct hump around $j = 10$. But SSR does not affect the lengths of the Gram-Schmidt vectors $\hat{\mathbf{b}}_j$ with, say, $j > 20$ at all.

The latter phenomenon is even more apparent in diagram (b) that uses a logarithmic scale. Here all lengths are normalized with respect to the length of the respective first base vector $\mathbf{b}_1$. Furthermore, the values are permuted by $\sigma_{u,\mathsf{B}}$ as defined in (2.9), i. e., the Gram-Schmidt vectors are essentially sorted in non-increasing length order. Thus, this diagram shows the quantities that actually determine the success probability in the sampling loop. For $j > 20$, the graphs are more or less parallel. Therefore, one needs to introduce a correction factor C into (GSA), $\|\hat{\mathbf{b}}_j\|^2 \approx Cq^{j-1}\|\mathbf{b}_1\|^2$, if it is to describe the reality well.

3.2. Trivial CSSS

The most trivial Check Search Space Size implementation $\mathsf{CSSS}_{\mathsf{triv}}$ ignores its arguments and always returns true. Choosing this implementation means "no matter how small the success probability, exhaustively search $S_{u_{\max},\mathsf{B}}$".

SSR will then always terminate in the inner loop after sampling in vain $2^{u_{\max}}$ vec-

tors. This might be a viable strategy for small or moderate $u_{\max}$. But if $u_{\max}$ is chosen such that the runtime of SSR is dominated by the sampling of lattice points rather than the recurrent LLL reduction and the computing time for sampling $2^{u_{\max}}$ vectors is already near the limit the user can spend, then waisting $2^{u_{\max}}$ samples really hurts. In that case a user most likely prefers to keep sampling only if there is a significant chance to improve the approximation factor. The CSSS implementations proposed in the following sections base their return value on estimations of the sampling's success probability.

3.3. CSSS Following Schnorr's Approach

As discussed above, Schnorr's analysis of RSR does no longer apply in general if we cannot presume (GSA). Nevertheless, we can numerically evaluate all relevant quantities and therefore estimate at runtime the success probability of a single sample following the idea of Schnorr. From this we infer an estimate for the minimum search space size that guarantees an at least 50% chance that a further SSR iteration will find a shorter vector.

In the following, we first state a formula for the mean of $\|\mathbf{v}\|^2$ taken over all $\mathbf{v} \in S_{u,\mathsf{B}}$ that satisfy $(\mathcal{E}_{k,u,\mathsf{B}})$. We then describe $\mathsf{CSSS}_{\mathsf{event}}$ that is based on the evaluation of that conditional mean value.

3.3.1. The expected length of v.

Lemma 12 stated a formula for $\Pr[(\mathcal{S}_{k,q_{\mathsf{B}}})]$ that facilitated the further analysis of RSR's inner loop under (GSA). We do not have a similar closed formula if B is not subject to (GSA). But (RA), equation (2.3), and the definition of $(\mathcal{E}_{k,u,r})$ yield a formula for the mean squared length of sampled vectors $\mathbf{v} \in S_{u,\mathsf{B}}$ under the precondition $(\mathcal{E}_{k,u,r})$. This formula can easily be evaluated numerically:

$$\begin{aligned} E(\mathsf{B}; k, u, r) :=& E\left[\, \|\mathbf{v}\|^2 \,\middle|\, (\mathcal{E}_{k,u,r}) \,\right] \\ =& \sum_{j=1}^{n} E\left[\, \nu_j^2 \|\hat{\mathbf{b}}_j\|^2 \,\middle|\, (\mathcal{E}_{k,u,r}) \,\right] = \sum_{j=1}^{n} s_j(k,u,r) \|\hat{\mathbf{b}}_{\sigma_{u,\mathsf{B}}(j)}\|^2 \end{aligned} \tag{3.1}$$

where

$$s_j(k,u,r) = \begin{cases} \frac{1}{12} r^{k-j} & \text{if } 1 \le j < k, \\ \frac{1}{12} & \text{if } k \le j < n-u, \\ \frac{1}{3} & \text{if } n-u \le j < n, \\ 1 & \text{if } j = n. \end{cases} \tag{3.2}$$

Note that we did not fix the parameters k, u, and r yet. Given B, we can choose them subject to the preconditions of Def. 11 and $u \le u_{\max}$.

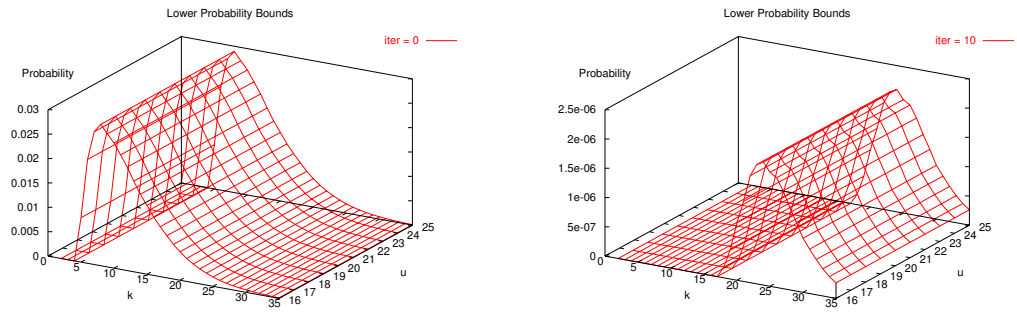


Figure 3.3.: Success Probability Bounds in $\mathsf{CSSS}_{\mathsf{event}}$

3.3.2. The $\mathsf{CSSS}_{\mathsf{event}}$ implementation.

In analogy to (2.8), we have

$$\Pr\left[\,\|\mathbf{v}\|^2 \le 0.99\|\mathbf{b}_1\|^2\,\right] \ge \tfrac{1}{2}\Pr\left[\,(\mathcal{E}_{k,u,r})\,\right] \tag{3.3}$$

provided $E(\mathsf{B}; k, u, r) \le 0.99\|\mathbf{b}_1\|^2$. The $\mathsf{CSSS}_{\mathsf{event}}$ algorithm – specified in Alg. 4 – is built around the function $\mathsf{LogSuccessProbBound}$. Based on (2.10), (3.1) and (3.3), this function computes a lower bound for $\log_2 \Pr\left[\,\|\mathbf{v}\|^2 \le 0.99\|\mathbf{b}_1\|^2\,\right]$ that depends – besides the Gram-Schmidt vectors – on the free parameters k and u. To be more precise, $\mathsf{LogSuccessProbBound}(\ell, k, u, \|\mathbf{b}_1\|^2)$ determines

$$\left\lfloor \max\{\, \log_2 \tfrac{1}{2} r^{\frac{k(k-1)}{4}} \mid r \in [0,1] \text{ and } E(\mathsf{B}; k, u, r) \le 0.99\|\mathbf{b}_1\|^2 \,\} \cup \{\, -\infty \,\} \right\rfloor .$$

Observe that $\Pr\left[\,(\mathcal{E}_{k,u,r})\,\right]$ is strictly increasing in r whence we need to find the maximum $r_{\max}$ such that the conditional mean value of $\|\mathbf{v}\|^2$ does not exceed $0.99\|\mathbf{b}_1\|^2$. On the other hand, $E(\mathsf{B}; k, u, r)$ is continuous and also strictly increasing in r. Therefore, $r_{\max}$ is the unique root $r_{\max} \in [0,1]$ of the equation $E(\mathsf{B}; k, u, r) = 0.99\|\mathbf{b}_1\|$ and, if it exists, then we can determine this root by, e. g., the textbook Regula Falsi method [PTVF92]. All that is left for $\mathsf{LogSuccessProbBound}$ is to check first for the special cases $\max\{\, E(\mathsf{B}; k, u, r) \mid r \in [0,1] \,\} \le 0.99\|\mathbf{b}_1\|^2$ as well as $\min\{\, E(\mathsf{B}; k, u, r) \mid r \in [0,1] \,\} \ge 0.99\|\mathbf{b}_1\|^2$ and to compute by (2.10) the actual probability estimate once $r_{\max}$ is known.

The evaluation of $E(\mathsf{B}; k, u, r)$ in $\mathsf{ExpLength}$ is straightforward. One has to take precautions, however, to avoid loss of floating point precision errors because the largest and the smallest $\|\hat{\mathbf{b}}_j\|$ often differ by several orders of magnitude. Therefore, one has to accumulate the summands in non-decreasing order.

For reasonable values of $u_{\max}$, say $u_{\max} \le 50$, the squared length of the orthogonalized vectors $\hat{\mathbf{b}}_j$, $n - u \le j \le n$, is typically very small compared to $\|\hat{\mathbf{b}}_j\|^2$ for small j. This is obvious under (GSA), but holds also in practice most of the time. Therefore, the value of u has not much impact on $E(\mathsf{B}; k, u, r)$ and in turn on the result of $\mathsf{LogSuccessProbBound}$. Fig. 3.3 exhibits this effect. The two diagrams show the success

Algorithm 4 $\mathsf{CSSS}_{\mathsf{event}}$

Input: • $\mathfrak{b}$: vector of squared Gram-Schmidt vector lengths
$\mathfrak{b} = (\|\hat{\mathbf{b}}_1\|^2, \ldots, \|\hat{\mathbf{b}}_n\|^2)$ where $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R} \in \mathbb{Z}^{d\times n}$, $n > 3$
• $u_{\max}$: $\log_2$ of maximum search space size

Output: true if and only if

$$\exists k \in \{1, \ldots, n - u_{\max} - 1\}, u \in \{1, \ldots, u_{\max}\}, r \in [0,1] :$$
$$E(\mathsf{B}; k, u, r) \le 0.99\|\mathbf{b}_1\|^2 \quad \text{and} \quad 2^{u_{\max}} \ge 1/\left(\tfrac{1}{2}\Pr\left[(\mathcal{E}_{k,u,r})\right]\right)$$

procedure $\mathsf{ExpLength}(\ell, k, u, r)$ /* $\ell = (l_1, \ldots, l_n)$ */
  return $\frac{1}{12}\sum_{i=1}^{k-1} r^{k-i} l_i + \frac{1}{12}\sum_{i=k}^{n-u-1} l_i + \frac{1}{3}\sum_{i=n-u}^{n-1} l_i + l_n$
end procedure

procedure $\mathsf{LogSuccessProbBound}(\ell, k, u, \|\mathbf{b}_1\|^2)$
  if $\mathsf{ExpLength}(\ell, k, u, 1) \le 0.99\|\mathbf{b}_1\|^2$ **then**
    return -1
  else if $\mathsf{ExpLength}(\ell, k, u, 0) \ge 0.99\|\mathbf{b}_1\|^2$ **then**
    return $-\infty$
  end if
  $r_{\max} \leftarrow \mathsf{RegulaFalsi}(\mathsf{ExpLength}(\ell, k, u, q) = 0.99\|\mathbf{b}_1\|^2, r \in [0,1])$
  return $\lfloor \frac{k(k-1)}{4}\log_2(r_{max}) - 1 \rfloor$
end procedure

procedure $\mathsf{CSSS}_{\mathsf{event}}(\mathfrak{b}, u_{\max})$
  for u **from** $u_{\max}$ **downto** 1 **do**
    $\ell \leftarrow (\|\hat{\mathbf{b}}_{\sigma_{u,\mathsf{B}}(1)}\|^2, \ldots, \|\hat{\mathbf{b}}_{\sigma_{u,\mathsf{B}}(n)}\|^2)$ /* $\sigma_{u,\mathsf{B}}$ defined by (2.9), page 21 */
    if $u_{\max} \ge -\max_{1\le k<n-u_{\max}} \mathsf{LogSuccessProbBound}(\ell, k, u, \|\mathbf{b}_1\|^2)$ **then**
      return true
    end if
  end for
  return false
end procedure

probability bounds $\frac{1}{2}\Pr\left[(\mathcal{E}_{k,u,r})\right]$ before the first and after the tenth iterations of SSR. (This particular example was computed from a public Micciancio key in dimension 180. All LLL-type reductions were performed with BKZ, $(\delta, \beta) = (0.99, 5)$.) The probability bounds are clearly dominated by k.

In consequence, if $\mathsf{CSSS}_{\mathsf{event}}$ returns true then we expect it to terminate in the first loop iteration with $u = u_{\max}$, except for corner cases. In order to avoid computations that are unlikely to yield a positive result anyway one might therefore decide to skip all loop iterations in $\mathsf{CSSS}_{\mathsf{event}}$ with $u < u_{\max}$ and return false if the success probability estimate with $u = u_{\max}$ is too small for all admissible k. However, the runtime of $\mathsf{CSSS}_{\mathsf{event}}$ is negligible compared to the sampling loop and the repeated LLL reduction, so the advantage of this optimization is debatable.

3.4. CSSS by Convolution

The success probability computed by Schnorr's approach under (GSA) and by the algorithm $\mathsf{CSSS}_{\mathsf{event}}$ without (GSA) is based on quite rough estimates. E. g., it neglects all the sufficiently short vectors that match no event ($\mathcal{E}_{k,u,r}$) such that $E(\mathsf{B}; k, u, r) \leq 0.99\|\mathbf{b}_1\|^2$. In fact, the average number of samples required in our experiments until a sufficiently short vector is found is orders of magnitude smaller than predicted by $\mathsf{CSSS}_{\mathsf{event}}$.

We present an algorithm $\mathsf{CSSS}_{\mathsf{Fourier}}$ that overcomes this problem and numerically determines the minimum number of samples required to guarantee under (RA) a success probability of SSR's sampling loop not less than $1/2$. The approach taken by $\mathsf{CSSS}_{\mathsf{Fourier}}$ is to determine $\Pr\left[\,\|\mathbf{v}\|^2 \leq x\,\right] = \lambda(\mathcal{B}_{n-1}(r(x)) \cap Q)/\lambda(Q)$ by means of the convolution theorem where $\mathcal{B}_{n-1}(r(x)) \subset \mathbb{R}^{n-1}$ is the ball with radius $r(x) := (x - \|\hat{\mathbf{b}}_n\|^2)^{1/2}$, $Q \subset \mathbb{R}^{n-1}$ is a suitable cuboid, and λ is the volume, i. e., the Lebesgue measure.

After fixing our notion of Fourier transform and introducing the Fresnel integral function, we will first give an exact formula for the distribution of $\|\mathbf{v}\|^2$ in terms of the inverse Fourier transform of a product of Fresnel integrals. However, a naïve evaluation of this formula by discrete Fourier transform will suffer from so called aliasing effects. Our algorithm $\mathsf{CSSS}_{\mathsf{Fourier}}$ in Sect. 3.4.3 will therefore counteract those aliasing effects.

3.4.1. Preliminaries

Even though Fourier transformation of complex valued functions is a well established concept in mathematics, the scaling factors involved are not generally agreed upon. For example, [BS89], [PTVF92], [FJ03], and [For84] all use definitions of the Fourier transform that differ in details. We therefore need to settle on one definition.

Definition 13. *Let $g, h : \mathbb{R} \to \mathbb{C}$ be Lebesgue integrable functions. Then*

$$\mathcal{F}\, g : \mathbb{R} \to \mathbb{C} : x \mapsto \int_{\mathbb{R}} g(y) e^{2\pi \imath xy}\, dy \tag{3.4}$$

$$\mathcal{F}^{-1}\, h : \mathbb{R} \to \mathbb{C} : x \mapsto \int_{\mathbb{R}} h(y) e^{-2\pi \imath xy}\, dy \tag{3.5}$$

defines the Fourier transform *of g and the* inverse Fourier transform *of h, respectively. The* convolution product *of g and h is*

$$g * h : \mathbb{R} \to \mathbb{C} : x \mapsto \int_{\mathbb{R}} g(y) h(x - y)\, dy\,. \tag{3.6}$$

In the following, we need only basic facts of the rich theory of Fourier transforms. We state them here for reference purposes.

Proposition 17. *Let $g, h : \mathbb{R} \to \mathbb{C}$ be Lebesgue integrable.*

a) Both $\mathcal{F}\, g$ and $\mathcal{F}^{-1}\, g$ are continuous and

$$\lim_{|y|\to\infty} (\mathcal{F}\, g)(y) = 0 = \lim_{|y|\to\infty} (\mathcal{F}^{-1}\, g)(y)\,.$$

b) g is a purely real function (up to some zero set) if and only if

$$(\mathcal{F}\, g)(-y) = \overline{(\mathcal{F}\, g)(y)}$$

for all $y \in \mathbb{R}$.

c) If $\mathcal{F}\, g$ is Lebesgue integrable as well then

$$\mathcal{F}^{-1}(\mathcal{F}\, g) = g = \mathcal{F}(\mathcal{F}^{-1}\, g)$$

almost everywhere.

d) ("Convolution Theorem") The convolution product $g * h$ is Lebesgue integrable and

$$\mathcal{F}(g * h) = (\mathcal{F}\, g) \cdot (\mathcal{F}\, h)\,.$$

e) If f, g are piecewise continously differentiable, then so is their convolution product and

$$\partial_x(f * g) = (\partial_x f) * g = f * (\partial_x g)\,.$$

Proof. E. g., [For84]. □

Let $X \in_R (-t, t]$ be a uniform random variable. As we will see below, the Fourier transform of the distribution of X^2 involves the so called Fresnel integral function.

Definition 14. *The function*

$$\mathrm{Fr} : \mathbb{R} \to \mathbb{C} : t \mapsto \int_0^t e^{2\pi\imath \frac{x^2}{4}}\, dx \tag{3.7}$$

is named Fresnel integral function.

The literature (e. g., [Wei05, PTVF92]) often refers to a pair $C : \mathbb{R} \to \mathbb{R}$ and $S : \mathbb{R} \to \mathbb{R}$ of Fresnel integrals where $\mathrm{Fr} = C + \imath S$. The Fresnel integral, which is closely related to the Gauss error function erfc, has applications in signal processing, optics, statistics, and even railway engineering, and its efficient computation is well studied. Its graph, depicted in Fig. 3.4, is known as Cornu spiral. The Fresnel integral is an odd function and

$$\mathrm{Fr}(t) \sim \tfrac{1+\imath}{2} + \tfrac{1}{\pi t} e^{2\pi\imath \frac{t^2-1}{4}} \tag{3.8}$$

for $t \gg 1$. In particular, Fr is bounded.

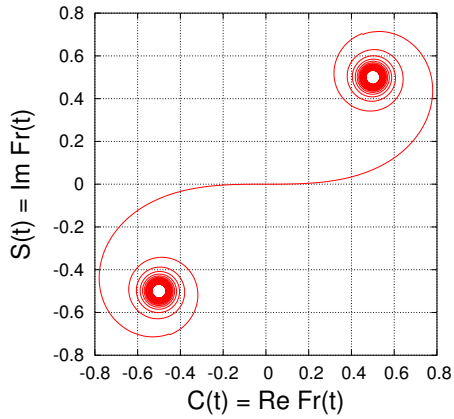


Figure 3.4.: The Cornu Spiral

3.4.2. The Probability Function of $\|\mathbf{v}\|^2$

We give an exact formula under (RA) for $\Pr\left[\,\|\mathbf{v}\|^2 \le t\,\right]$, $\mathbf{v} \in_R S_{u,\mathsf{B}}$, in terms of the Fourier transform of its distribution function. For this, we consider the intersection of the n-dimensional ball $\mathcal{B}_n(x) = \{\,\mathbf{v} \in \mathbb{R}^n \mid \|\mathbf{v}\| \le x\,\}$ with an n-dimensional cuboid $Q_{t_1,\ldots,t_n} = (-t_1, t1] \times \cdots \times (-t_n, t_n]$, $t_j > 0$, both centered at the origin. We determine the ratio $R_{t_1,\ldots,t_n}(x)$ of vectors in $Q_{t_1,\ldots,t_n}$ that are shorter than $\sqrt{x}$,

$$R_{t_1,\ldots,t_n} : \mathbb{R} \to \mathbb{R} : x \mapsto \begin{cases} \frac{\lambda\left(\mathcal{B}_n\left(\sqrt{x}\right) \cap Q_{t_1,\ldots,t_n}\right)}{\lambda\left(Q_{t_1,\ldots,t_n}\right)} & \text{for } x \ge 0, \\ 0 & \text{for } x < 0. \end{cases} \tag{3.9}$$

λ denominates the Lebesgue measure. We write ∂_x as a shorthand for the differential operator $\frac{d}{dx}$.

Theorem 18. *With the notation above and $n > 2$, define $\varphi_{t_1,\ldots,t_n} : \mathbb{R} \to \mathbb{C}$ as the unique continuous function such that, for $y > 0$,*

$$\varphi_{t_1,\ldots,t_n}(y) = \prod_{j=1}^{n} \frac{\mathrm{Fr}(2t_j\sqrt{y})}{2t_j\sqrt{y}} \quad \textit{and} \quad \varphi_{t_1,\ldots,t_n}(-y) = \overline{\varphi_{t_1,\ldots,t_n}(y)}\,. \tag{3.10}$$

In particular, $\varphi_{t_1,\ldots,t_n}(0) = 1$. Then $R_{t_1,\ldots,t_n} \in C^1(\mathbb{R})$ and

$$R_{t_1,\ldots,t_n}(x) = \int_0^x (\mathcal{F}^{-1}\,\varphi_{t_1,\ldots,t_n})(s)\,ds\,. \tag{3.11}$$

Before we set out to proof Theorem 18, let us apply it to our problem at hand:

Corollary 19. *Let $\mathsf{B} \in \mathbb{Z}^{d\times n}$, $n > 2$, be a lattice basis with Gram-Schmidt decomposition $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$ and $1 \le u < n$. Let $\mathbf{v} = \sum_{j=1}^{n} \nu_j \hat{\mathbf{b}}_j \in_R S_{u,\mathsf{B}}$ be uniformly sampled. For $1 \le j < n$ set $t_j = \|\hat{\mathbf{b}}_j\|$ if $n - u \le j < n$ and $t_j = \frac{1}{2}\|\hat{\mathbf{b}}_j\|$ else.*

Then, under (RA), $D_{\|\mathbf{v}\|^2} : \mathbb{R} \to \mathbb{R} : x \mapsto (\mathcal{F}^{-1}\varphi_{t_1,\dots,t_{n-1}})(x - \|\hat{\mathbf{b}}_n\|^2)$ *is the distribution function of* $\|\mathbf{v}\|^2$ *where* $\varphi_{t_1,\dots,t_{n-1}}$ *is defined as in Theorem 18 and*

$$\Pr\left[\,\|\mathbf{v}\|^2 \le x\,\right] = \int_0^{x-\|\hat{\mathbf{b}}_n\|^2} (\mathcal{F}^{-1}\varphi_{t_1,\dots,t_{n-1}})(s)\,ds\,. \tag{3.12}$$

Proof. According to (RA) and the definition of $S_{u,\mathsf{B}}$, the random function $\mathbf{u} := \mathbf{v} - \hat{\mathbf{b}}_n \in Q_{t_1,\dots,t_{n-1}}$ is uniformly distributed. This implies

$$\Pr\left[\,\|\mathbf{v}\|^2 \le x\,\right] = \Pr\left[\,\|\mathbf{u}\|^2 \le x - \|\hat{\mathbf{b}}_n\|^2\,\right] = R_{t_1,\dots,t_{n-1}}(x - \|\hat{\mathbf{b}}_n\|^2)$$

because $\|\mathbf{u}\|^2 \le x - \|\hat{\mathbf{b}}_n\|^2$ if and only if $\mathbf{u} \in \mathcal{B}_{n-1}\Big((x - \|\hat{\mathbf{b}}_n\|^2)^{1/2}\Big)$. □

We prepare the proof of Theorem 18 by computing the Fourier transform $\mathcal{F}\,\partial_x R_t$ of the 1-dimensional ratio's derivative. Afterward, we observe that the n-dimensional ratio's derivative is the convolution of the derivatives of the respective 1-dimensional ratios. Then everything falls in place due to the convolution theorem.

First, we make the formulas for the case $n = 1$ explicit:

Lemma 20. *Let* $R_t : \mathbb{R} \to \mathbb{R}$, $t > 0$, *as above. Then, for all* $x \in \mathbb{R} \setminus \{\,0, t^2\,\}$,

$$R_t(x) = \begin{cases} 0 & \text{if } x < 0, \\ \frac{\sqrt{x}}{t} & \text{if } 0 < x < t^2, \\ 1 & \text{if } t^2 < x, \end{cases} \quad \text{and} \quad \partial_x R_t(x) = \begin{cases} \frac{1}{2t\sqrt{x}} & \text{if } 0 < x < t^2, \\ 0 & \text{else.} \end{cases} \tag{3.13}$$

Furthermore, $\partial_x R_t$ *is Lebesgue integrable,* $(\mathcal{F}\,\partial_x R_t)(0) = 1$, *and, for all* $y > 0$,

$$(\mathcal{F}\,\partial_x R_t)(y) = \frac{\mathrm{Fr}(2t\sqrt{y})}{2t\sqrt{y}} \quad \text{and} \quad (\mathcal{F}\,\partial_x R_t)(-y) = \frac{\overline{\mathrm{Fr}(2t\sqrt{y})}}{2t\sqrt{y}}\,. \tag{3.14}$$

Proof. The 1-dimensional ball $\mathcal{B}_1(\sqrt{x}) = [-\sqrt{x}, \sqrt{x}]$ and the cuboid $Q_t = (-t, t]$ are both in fact intervals in $\mathbb{R}$ and therefore $\mathbb{R}_t(x) = \lambda\left([-\sqrt{x}, \sqrt{x}]\right)/\lambda\left((-t, t]\right) = \sqrt{x}/t$ if $0 \le x \le t^2$. For larger x the ratio is saturated because of $(-t, t] \subseteq [-\sqrt{x}, \sqrt{x}]$. The formula for $\partial_x R_t(x)$ follows immediately by elementary derivation.

$\partial_x R_t$ is Riemann integrable on $\mathbb{R} \setminus \{\,0, t^2\,\}$ by the second fundamental theorem of calculus and therefore Lebesgue integrable on $\mathbb{R}$. Thus, we can apply the Fourier operator. By Prop. 17, $\mathcal{F}\,\partial_x R_t$ has the Hermitian property $(\mathcal{F}\,\partial_x R_t)(-y) = \overline{(\mathcal{F}\,\partial_x R_t)(y)}$ and is continuous. Now assume $y > 0$. Then

$$\begin{aligned}(\mathcal{F}\,\partial_x R_t)(y) &= \int_{\mathbb{R}} \partial_x R_t(z) e^{2\pi\imath yz}\,dz = \int_0^{t^2} \frac{1}{2t\sqrt{z}} e^{2\pi\imath yz}\,dz \\ &= \int_0^{2\sqrt{y}t} \frac{2\sqrt{y}}{2tx} e^{2\pi\imath\frac{x^2}{4}}\,\frac{x}{2y}dx = \frac{1}{2t\sqrt{y}}\int_0^{2t\sqrt{y}} e^{2\pi\imath\frac{x^2}{4}}\,dx = \frac{\mathrm{Fr}(2t\sqrt{y})}{2t\sqrt{y}}\end{aligned}$$

by means of the substitution $x(z) = 2\sqrt{y}\sqrt{z}$. In particular,

$$(\mathcal{F}\,\partial_x R_t)(0) = \lim_{y\to 0} \frac{\mathrm{Fr}(2t\sqrt{y})}{2t\sqrt{y}} = \lim_{x\to 0} \frac{\mathrm{Fr}(x) - \mathrm{Fr}(0)}{x} = (\partial_x \mathrm{Fr})(0) = 1$$

since $\mathrm{Fr}(0) = 0$. □

Lemma 21. *Let $n \geq 1$ and $R_{t_1,\dots,t_n}$, $t_j > 0$, as above. Then*

$$R_{t_1,\dots,t_n}(x) = \int_0^x \left(\partial_x R_{t_1} * \cdots * \partial_x R_{t_n}\right)(s)\, ds \tag{3.15}$$

for all $x \in \mathbb{R}$. $R_{t_1,\dots,t_n}$ is at least piecewise continuously differentiable.

Proof. We show $\partial_x R_{t_1,\dots,t_n} = \partial_x R_{t_1} * \cdots * \partial_x R_{t_n}$ by induction. The lemma is trivially true if $n = 1$.

Now assume the lemma holds for $n \geq 1$. Set $M(x) := \mathcal{B}_n(x) \cap Q_{t_1,\dots,t_n}$ and $N(x) := \mathcal{B}_{n+1}(x) \cap Q_{t_1,\dots,t_{n+1}}$. We can, by definition, express the Lebesgue measure of a set as integral over the corresponding characteristic function, i. e., $\lambda(N(\sqrt{x})) = \int_{\mathbb{R}^{n+1}} \chi_{N(\sqrt{x})}(\mathbf{s})\, d\mathbf{s}$. With $\mathbf{s} = (\mathbf{s}', s_{n+1}) \in \mathbb{R}^n \times \mathbb{R}$, we have $\mathbf{s} \in N(\sqrt{x})$ if and only if $\mathbf{s}' \in M(\sqrt{x - s_{n+1}^2})$. Applying Fubini's lemma, we find

$$\begin{aligned}
R_{t_1,\dots,t_{n+1}}(x) &= 2\int_0^{t_{n+1}} \frac{\lambda\left(M(\sqrt{x-s^2})\right)}{\lambda\left(Q_{t_1,\dots,t_n}\right)} \frac{1}{\lambda\left(Q_{t_{n+1}}\right)}\, ds \\
&= 2\int_0^{t_{n+1}^2} \frac{\lambda\left(M(\sqrt{x-y})\right)}{\lambda\left(Q_{t_1,\dots,t_n}\right)} \frac{1}{2t_{n+1}} \frac{dy}{2\sqrt{y}} \\
&= \int_{\mathbb{R}} R_{t_1,\dots,t_n}(x-y) \cdot \partial_x R_{t_{n+1}}(y)\, dy \\
&= \left(R_{t_1,\dots,t_n} * \partial_x R_{t_{n+1}}\right)(x)
\end{aligned}$$

by substituting $y(s) = s^2$ and Lemma 20. Since both $\partial_x R_{t_{n+1}}$ and, by assumption, $R_{t_1,\dots,t_n}$ are piecewise continously differentiable, we have

$$\partial_x R_{t_1,\dots,t_{n+1}} = \partial_x R_{t_1,\dots,t_n} * \partial_x R_{t_{n+1}} = \partial_x R_{t_1} * \cdots * \partial_x R_{t_n} * \partial_x R_{t_{n+1}}\,.$$

by Prop. 17, i. e., the lemma also holds for $n + 1$. □

Lemma 22. *Let $\varphi_{t_1,\dots,t_n}$ as above. Then $|\varphi_{t_1,\dots,t_n}(y)| \in O(|y|^{-n/2})$. If $n > 2$ then $\varphi_{t_1,\dots,t_n}$ is Lebesgue integrable.*

Proof. We know $\mathrm{Fr}(y) \sim \mathrm{sgn}(y)(\frac{1+\imath}{2} + \frac{1}{\pi|y|} e^{2\pi\imath \frac{y^2-1}{4}})$ for large $|y|$, so $|\mathrm{Fr}(2t\sqrt{|y|})| \in O(1)$ for any $t > 0$. Therefore,

$$|\varphi_{t_1,\dots,t_n}(y)| \in O\big(\textstyle\prod_{j=1}^n \frac{1}{2t_j\sqrt{|y|}}\big) = O\big(|y|^{-n/2}\big)\,.$$

The function $f_c : \mathbb{R} \to \mathbb{R} : y \mapsto c|y|^{-n/2}$ is Lebesgue integrable for all $n > 2$ and all $c \in \mathbb{R}$. Since $|\varphi_{t_1,\dots,t_n}|$ is continuous and there is $c \in \mathbb{R}$ such that $|\varphi_{t_1,\dots,t_n}| \leq f_c$, $\varphi_{t_1,\dots,t_n}$ must be Lebesgue integrable as well. □

Proof of Theorem 18. According to Lemma 20, $\partial_x R_{t_j}$ is Lebesgue integrable for all $j \in \{1, \dots, n\}$. We know from Prop. 17 and Lemma 21 that their convolution product $\partial_x R_{t_1,\dots,t_n} = \partial_x R_{t_1} * \cdots * \partial_x R_{t_n}$ is Lebesgue integrable as well. By the convolution theorem, Lemma 20, and the definition of $\varphi_{t_1,\dots,t_n}$,

$$\mathcal{F}\,\partial_x R_{t_1,\dots,t_n} = \mathcal{F}\,\partial_x R_{t_1} \cdot \ldots \cdot \mathcal{F}\,\partial_x R_{t_n} = \varphi_{t_1,\dots,t_n}\,.$$

Since both $\partial_x R_{t_1,\dots,t_n}$ and $\varphi_{t_1,\dots,t_n}$ are Lebesgue integrable (Lemma 22), Prop. 17 guarantees

$$\partial_x R_{t_1,\dots,t_n} = \mathcal{F}^{-1}\,\mathcal{F}\,\partial_x R_{t_1,\dots,t_n} = \mathcal{F}^{-1}\,\varphi_{t_1,\dots,t_n}$$

almost everywhere. The left hand side is piecewise continuous by Lemma 21, the right hand side is continuous by Prop.17. Therefore, the functions must be equal everywhere and $R_{t_1,\dots,t_n} \in C^1(\mathbb{R})$. □

Remark 4. For completeness' sake, we also give a formula for $\partial_x R_{t_1,\dots,t_n}$ if $n = 2$. W. l. o. g., assume $0 < t_1 \le t_2$. Using entry no. 146 from the integral tables in [BS89], we find

$$\begin{aligned}
\partial_x R_{t_1,t_2}(x) &= \left[\frac{\operatorname{sgn}(s)}{2t_1t_2}\arctan\left(\sqrt{\frac{s}{x-s}}\right)\right]_{\max\{0,x-t_2\}}^{\min\{x,t_1\}} \\
&= \begin{cases}
\frac{\pi}{4t_1t_2} & \text{if } 0 < x \le t_1^2, \\
\frac{\pi - 2\arctan\left(\sqrt{t_1^2/(x-t_1^2)}\right)}{4t_1t_2} & \text{if } t_1^2 < x \le t_2^2, \\
\frac{\arctan\left(\sqrt{(x-t_2^2)/t_2^2}\right) - \arctan\left(\sqrt{t_1^2/(x-t_1^2)}\right)}{2t_1t_2} & \text{if } t_2^2 < x \le t_1^2 + t_2^2, \\
0 & \text{otherwise.}
\end{cases}
\end{aligned} \tag{3.16}$$

$\partial_x R_{t_1,t_2}$ is continuous except in $x = 0$.

3.4.3. The $\mathsf{CSSS}_{\mathsf{Fourier}}$ Algorithm

In the following, we propose a Check Search Space Size implementation $\mathsf{CSSS}_{\mathsf{Fourier}}$, Alg. 5, that is based on Cor. 19. Our algorithm computes the inverse Fourier transform by means of the inverse discrete Fourier transform. It avoids the significant errors caused by the so called aliasing effect that a naïve application of the inverse discrete Fourier transform would incur.

As suggested by Cor. 19, most of the work is delegated to a function Ratio that computes $R_{t_1,\dots,t_m}(x)$, $m > 2$. $\mathsf{CSSS}_{\mathsf{Fourier}}$ first determines the parameters t_j for $j \in \{1, \dots, n-1\}$ and then computes $p := \Pr\left[\,\|\mathbf{v}\|^2 \le 0.99\|\mathbf{b}_1\|\,\right] = R_{t_1,\dots,t_{n-1}}(0.99)$. It finally tests whether the probability that $S_{u,\mathsf{B}}$ does not contain a sufficiently short vector is $1/2$ or less, i. e., $(1-p)^{2^u} \le 1/2$ or equivalently $\log_2(1-p) \le -2^{-u}$. Even if p is very small, $\log_2(1-p)$ can be efficiently computed with sufficiently high precision by a power series expansion.

Algorithm 5 $\mathsf{CSSS}_{\mathsf{Fourier}}$

Input:
- $\mathfrak{b}$: vector of squared Gram-Schmidt vector lengths $\mathfrak{b} = (\|\hat{\mathbf{b}}_1\|^2, \ldots, \|\hat{\mathbf{b}}_n\|^2)$ where $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R} \in \mathbb{Z}^{d \times n}$, $n > 3$
- u: $\log_2$ of maximum search space size
- D: global parameter, 2^D DFT sample points

Output: true if and only if $\Pr\left[\, \|\mathbf{v}\|^2 \leq 0.99\|\mathbf{b}_1\|^2 \,\right] > 2^{1-u_{\max}}$

```
procedure RDeriv ((t_1, ..., t_m), T, N)        /* T > 2√(t_1^2 + ··· + t_m^2), N > 1 */
    for j from 0 to N/2 do
        (Φ_j, k) ← (φ_{t_1,...,t_m}(j/T), 0)
        repeat
            δ_Φ ← φ_{t_1,...,t_m}((j + kN)/T) + φ_{t_1,...,t_m}((j − kN)/T)
            (Φ_j, k) ← (Φ_j + δ_Φ, k + 1)
        until δ_Φ < ε                    /* equality up to floating point precision */
    end for
    return (N/(4T)) DFT^{-1}((Φ_0, ..., Φ_{N/2}, \overline{Φ_{N/2−1}}, ..., \overline{Φ_1})^t)
end procedure

procedure Ratio (x, (t_1, ..., t_m))
    N ← 2^D
    T ← min{ 2^τ | τ ∈ Z and 2^τ > 2√(t_1^2 + ··· + t_m^2) }
    (d_0, ..., d_{N−1})^t ← RDeriv((t_1, ..., t_m), T, N)
    return Quadrature(x, (d_0, ..., d_{N−1}), T, N)
end procedure

procedure CSSS_Fourier (𝔟, u)
    (t_1, ..., t_{n−1}) ← (½‖b̂_1‖, ..., ½‖b̂_{n−u−1}‖, ‖b̂_{n−u}‖, ..., ‖b̂_{n−1}‖)/‖b̂_1‖
    p ← Ratio(0.99 − ‖b̂_n‖^2/‖b_1‖^2, (t_1, ..., t_{n−1}))
    return log_2(1 − p) ≤ −2^{−u}
end procedure
```

Ratio asks yet another function RDeriv for the values of $(\partial_x R_{t_1,\ldots,t_m})(jT/N)$, $j \in \{\, 0, \ldots, N-1 \,\}$. The number N of sample points is set to a power of two in order to make it possible to choose the most efficient inverse discrete Fourier transform implementation in RDeriv. The exponent D is a global constant, typically $10 \leq D \leq 15$. We choose T as a power of 2 as well so the floating point representations of the sampling points jT/N are exact. The lower bound $T > 2\sqrt{t_1^2 + \cdots + t_m^2}$ is a requirement of RDeriv. The value of $R_{t_1,\ldots,t_m}(x)$ can then be recovered from the samples $(\partial_x R_{t_1,\ldots,t_m})(jT/N)$ by a standard quadrature formula, e. g., Simpson's rule [PTVF92]. Note that quadrature formulas approximate the integral function by piecewise polynomial functions. If $jT/N \leq x < (j+1)T/N$ then we approximate the integral up to jT/N using the quadrature formula and the integral from jT/N up to x by evaluating the interpolation polynomial used by the quadrature formula.

We omit here the details that can be looked up in any introductory textbook on numerical analysis.

Finally, the function RDeriv computes the vector $(\Phi_0, \dots, \Phi_{N/2}, \overline{\Phi_{N/2-1}}, \dots, \overline{\Phi_1})$ where Φ_j is the value of the absolutely convergent series $\sum_{k\in\mathbb{Z}} \varphi_{t_1,\dots,t_m}((j+kN)/T)$. The subsequent inverse discrete Fourier transformation of this vector reveals the samples $\partial_x R_{t_1,\dots,t_m}(jT/N)$, $j \in \{0, \dots, N-1\}$. The correctness of RDeriv under the preconditions stated in the comment is due to Lemma 23 which we will show below.

Remark 5. In order to keep the algorithm simple, RDeriv uses a naïve method to compute Φ_j. In general, numerical analysists frown upon the evaluation of a series the terms of which converge only polynomially to zero. However, sampling reduction is intended for use with high dimensional lattices, say $n > 100$. Then $|\varphi_{t_1,\dots,t_{n-1}}| \in O(|y|^{-(n-1)/2}) \subseteq O(|y|^{-50})$ becomes quickly smaller than the floating point precision, anyway. Still, we can do better: [PTVF92] describes a method due to Van Wijngaarden that can speed up the computation of Φ_j.

Discrete Fourier Transform and Aliasing. The discrete Fourier transform makes it possible to approach the Fourier transform numerically.

Definition 15. *Let $N \in \mathbb{N}$. The function*

$$\mathrm{DFT} : \mathbb{C}^N \to \mathbb{C}^N : (g_1, \dots, g_N)^t \mapsto (G_1, \dots, G_N)^t, \quad G_j = \sum_{k=0}^{N-1} g_k e^{2\pi\imath \frac{kj}{N}}, \tag{3.17}$$

is named discrete Fourier transform.

The discrete Fourier transform is an automorphism of $\mathbb{C}^N$ with inverse

$$\mathrm{DFT}^{-1} : \mathbb{C}^N \to \mathbb{C}^N : (G_1, \dots, G_N)^t \mapsto (g_1, \dots, g_N)^t, \quad g_j = \frac{1}{N} \sum_{k=0}^{N-1} G_k e^{-2\pi\imath \frac{kj}{N}}. \tag{3.18}$$

The DFT can be considered a discretization of integral (3.4) in N equidistant sample points: Let $g : \mathbb{R} \to \mathbb{C}$, $N \in \mathbb{N}$, and $\Delta > 0$ such that $\operatorname{supp} g \subseteq [0, N\Delta]$. Then, for $j \in \{0, \dots, N-1\}$,

$$\begin{aligned}(\mathcal{F} g)(\tfrac{j}{N\Delta}) &= \int_{\mathbb{R}} g(x) e^{2\pi\imath \frac{j}{N\Delta} x}\, dx \\ &\approx \sum_{k=0}^{N-1} g(k\Delta) e^{2\pi\imath \frac{j}{N\Delta} k\Delta}\, \Delta = \Delta \sum_{k=0}^{N-1} g(k\Delta) e^{2\pi\imath \frac{kj}{N}}.\end{aligned} \tag{3.19}$$

The DFT has many important applications in, e. g., signal processing. Implementations of highly optimized and numerically very stable DFT algorithms are freely available [FJ03]. These algorithms typically run in $O(N \log N)$ time and achieve an L_2 error $O(\sqrt{\log N})$ [FJ05].

If g is bandwidth limited to frequencies smaller in absolute value than the *Nyquist critical frequency* $f_\Delta = \frac{1}{2\Delta}$, i.e., $(\mathcal{F}g)(y) = 0$ for $|y| \geq f_\Delta$, then one can usually expect that the approximation (3.19) is quite good. However, if g is not bandwidth limited then *aliasing* can add significant errors. Aliasing refers to the inevitable phenomenon that the DFT value G_j, which is supposed to approximate $(\mathcal{F}g)(\frac{j}{N\Delta})$, in fact contains "power" from all $(\mathcal{F}g)(\frac{j+kN}{N\Delta})$, $k \in \mathbb{Z}$. This is a consequence of the fact that – according to Parseval's theorem – the total powers of both, a function $g : \mathbb{R} \to \mathbb{C}$ and a finite sequence $(g_1, \ldots, g_{N-1}) \in \mathbb{C}^N$,

$$\|g\|_2^2 = \|\mathcal{F}g\|_2^2, \qquad \|(g_0, \ldots, g_{N-1})^t\|_2^2 = \|\operatorname{DFT}(g_0, \ldots, g_{N-1})^t\|_2^2 \tag{3.20}$$

are invariant under continuous and discrete Fourier transform, respectively.

Since $|\varphi_{t_1,\ldots,t_n}(y)| > 0$ for all $y \neq 0$, $\partial_x R_{t_1,\ldots,t_n}$ is not bandwidth limited at all. Direct application of the inverse discrete Fourier transform to samples of $\varphi_{t_1,\ldots,t_n}$ is unlikely to result in good approximations of samples of $\partial_x R_{t_1,\ldots,t_n}$. We show in the following how we can compute the discrete Fourier transform of $\partial_x R_{t_1,\ldots,t_n}$ up to arbitrary precision, including aliasing, from $\varphi_{t_1,\ldots,t_n}$. Then the inverse discrete Fourier transform will recover the samples of $\partial_x R_{t_1,\ldots,t_n}$. That is, we forestall the effect of aliasing.

Fourier Series Expansion. The discrete Fourier transform is more or less the evaluation of a Fourier series with N coefficients. The following lemma – that shows the correctness of RDeriv in Alg. 5 – uses this to our advantage.

Lemma 23. *For $n > 2$, let $R_{t_1,\ldots,t_n}$ and $\varphi_{t_1,\ldots,t_n}$ as above and $T \in \mathbb{R}$ such that $T > 2\sqrt{t_1^2 + \cdots + t_n^2}$. For any positive $N \in \mathbb{N}$ set $\Delta = T/N$ and $r_j = (\partial_x R_{t_1,\ldots,t_n})(j\Delta)$, $j \in \{0, \ldots, N-1\}$. Then*

$$(r_0, \ldots, r_{N-1})^t = \operatorname{DFT}^{-1}((\Phi_0, \ldots, \Phi_{N/2}, \overline{\Phi_{N/2-1}}, \ldots, \overline{\Phi_1})^t) \tag{3.21}$$

where

$$\Phi_j = \tfrac{1}{4\Delta} \sum_{k\in\mathbb{Z}} \varphi_{t_1,\ldots,t_n}\left(\tfrac{j+kN}{T}\right). \tag{3.22}$$

Proof. The series $\Phi_j = \frac{1}{4\Delta}\sum_{k\in\mathbb{Z}} \varphi_{t_1,\ldots,t_n}\left(\frac{j+kN}{T}\right)$ converges absolutely due to Lemma 22; in particular, the ordering of its terms does not matter.

Let $f \in C(\mathbb{R})$ be square integrable on the interval $[-1/2, 1/2]$ and $f(-1/2) = f(1/2)$. Then f has the Fourier series expansion [BS89]

$$f(x) = \sum_{k\in\mathbb{Z}} c_k(f) e^{-2\pi\imath kx} \quad \text{for } -1/2 \leq x \leq 1/2, \qquad c_k(f) = \tfrac{1}{4}\int_{-1/2}^{1/2} f(y) e^{2\pi\imath ky}\, dy\,.$$

In particular, if $\operatorname{supp} f \subseteq [-1/2, 1/2]$, then $c_k(f) = \frac{1}{4}(\mathcal{F}f)(k)$.

Define $g := (\partial_x R_{t_1,\ldots,t_n}) \circ (T\,\mathrm{id})$. g is continuous with $\operatorname{supp} g \subseteq [-1/2, 1/2]$ and $g\,(j/N) = g\,((j-N)/N)$ if $N/2 \leq j < N$. Since g is Riemann integrable, it is also square integrable. Then

$$c_k(g) = \tfrac{1}{4}(\mathcal{F}\,g)(k) = \tfrac{1}{4T}(\mathcal{F}\,\partial_x R_{t_1,\ldots,t_n})\left(\tfrac{k}{T}\right) = \tfrac{1}{4T}\varphi_{t_1,\ldots,t_n}\left(\tfrac{k}{T}\right)$$

and, observing $\Phi_l = \frac{1}{4\Delta}\sum_{k\in\mathbb{Z}} \varphi_{t_1,\ldots,t_n}\left(\frac{l+kN}{T}\right) = N\sum_{k\in\mathbb{Z}} c_{l+kN}(g)$,

$$(\partial_x R_{t_1,\ldots,t_n})(j\Delta) = g\left(\tfrac{j}{N}\right) = \sum_{k\in\mathbb{Z}} c_k(g) e^{-2\pi\imath\frac{kj}{N}} = \tfrac{1}{N}\sum_{l=0}^{N-1} \Phi_l e^{-2\pi\imath\frac{lj}{N}}\,.$$

We find by comparison of the last equation with (3.18) – and with the shorthand $r_j = (\partial_x R_{t_1,\ldots,t_n})(j\Delta)$ – that $(r_0,\ldots,r_{N-1})^t = \mathrm{DFT}^{-1}((\Phi_0,\ldots,\Phi_{N-1})^t)$. Since all $r_j \in \mathbb{R}$, the Φ_j have the Hermitian property $\Phi_j = \overline{\Phi_{N-j}}$. □

Evaluation of the Fresnel integral. Above we always assumed that we can easily evaluate the Fresnel integral and therefore $\varphi_{t_1,\ldots,t_m}$. This is in fact the case due to the following: By element-wise integration of the exponential function's power series we find the power series

$$\mathrm{Fr}(x) = \sum_{j=0}^{\infty} \left(\frac{\pi\imath}{2}\right)^j \frac{x^{2j+1}}{(2j+1)\cdot j!} \tag{3.23}$$

that converges rapidly for small x. For large x, we write the Fresnel integral in terms of the Gauss error function

$$\mathrm{Fr}(x) = \frac{1+\imath}{2}\,\mathrm{erfc}(z), \quad z = \frac{\sqrt{\pi}}{2}(1-\imath)x \tag{3.24}$$

that can be evaluated by the continued fraction expansion

$$\mathrm{erfc}(z) = \frac{1}{\sqrt{\pi}} e^{-z^2} \cfrac{1}{2z^2 + 1 - \cfrac{1\cdot 2}{2z^2 + 5 - \cfrac{3\cdot 4}{2z^2 + 9 - \cdots}}}\,. \tag{3.25}$$

Based on these equations, [PTVF92] gives an algorithm Fresnel that efficiently computes the Fresnel integral function up to arbitrary precision.

3.5. Pool Sampling Reduction

We propose in Sect. 3.5.1 a generalization of SSR that exploits fairly short sample vectors to improve the overall quality of the result lattice basis. In Sect. 3.5.2 we describe how different pool sizes and acceptor parameters affect the algorithm's runtime and reduction result.

3.5.1. The PoolSR Algorithm and Randomized Acceptors

RSR and SSR concentrate on reducing the length of the first base vector. In fact, they typically do significantly reduce the length of the very first base vectors only. We propose a Sampling Reduction variant that generates more short vectors in the result basis.

Before the sampling loop in SSR finds a vector that is actually shorter than $\sqrt{0.99}\|\mathbf{b}_1\|$ it is likely to generate lattice points $\mathbf{v}$ such that $\|\mathbf{v}\| \approx \|\mathbf{b}_1\|$. SSR discards those vectors even though they are shorter than most of the base vectors. The *Pool Sampling Reduction* algorithm (PoolSR) stores some of the shortest sampled vectors and includes them in the generating system the LLL reduction of which becomes the lattice basis in the next iteration of the outer loop. The underlying idea is that we hope to improve the overall reduction result if we insert lattice points $\mathbf{v}$ into the generating system such that $\|\mathbf{v}\|^2 \ll \frac{1}{n}\sum_{j=1}^{n}\|\mathbf{b}_j\|^2$.

Besides the input basis, the maximum search space size, and the LLL reduction parameter, PoolSR (Alg. 6) expects a pool size $s \geq 1$ as well as some (possibly randomized) acceptor algorithm $\mathcal{A}$ that, on input a lattice vector $\mathbf{v}$, returns true or false. We suggest some reasonable acceptors below.

PoolSR maintains a "pool" $[\mathbf{p}_1, \ldots, \mathbf{p}_t]$ of at most s vectors. The pool vectors are stored in non-decreasing length order. Every time a vector $\mathbf{v}$ is sampled that is not short enough to terminate the sampling loop, $\mathbf{v}$ is passed to the acceptor algorithm $\mathcal{A}$. If $\mathcal{A}$ returns true or $\mathbf{v}$ is sufficiently short then $\mathbf{v}$ is inserted in the pool according to its length. If the pool already held s vectors then the longest pool vector is dropped whence the pool has never more than s elements. Besides $\mathbf{v}$, $\mathcal{A}$ may access all variables that define the current state of PoolSR, i. e., B, R, $\mathfrak{b}$, $u_{\max}$, and x. We do not write them as explicit parameters to keep the notation simple.

As in SSR, if a vector $\mathbf{v}$ is found such that $\|\mathbf{v}\|^2 \leq 0.99\|\mathbf{b}_1\|^2$ then the algorithm leaves the sampling loop and LLL reduces a new generating system of the lattice L. The only difference is that PoolSR inserts the remaining pool vectors in the generating system between $\mathbf{v} = \mathbf{p}_1$ and the base vectors $\mathbf{b}_j$, $j = 1, \ldots, n$.

The separate acceptor provides the flexibility to choose between different pool selection policies. For example, if $\mathcal{A} = \mathcal{A}_{\mathsf{SSR}}$ with

$$\mathcal{A}_{\mathsf{SSR}}(\mathbf{v}) = \text{false} \quad \text{for all } \mathbf{v} \in L(\mathsf{B}), \tag{3.26}$$

or if $s = 1$, then PoolSR becomes SSR. Thus, PoolSR is in fact a generalization of SSR. The other trivial acceptor $\mathcal{A}_{\text{all}}$ such that

$$\mathcal{A}_{\text{all}}(\mathbf{v}) = \text{true} \quad \text{for all } \mathbf{v} \in L(\mathsf{B}) \tag{3.27}$$

adds the s shortest vectors that were encountered in the sampling loop to the sampling loop. Using $\mathcal{A}_{\text{all}}$ can result in pool vectors that are significantly longer than $\mathbf{b}_1$ and therefore hinder rather than improve the reduction of B. This can be avoided by accepting vectors only if their length does not exceed a certain limit. Such a

Algorithm 6 PoolSR

Input:
- basis $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d \times n}$ of lattice L
- search space bound $u_{\max} \in \mathbb{N}$
- LLL parameter $\delta \in (1/4, 1)$
- pool size $s \geq 1$
- (randomized) acceptor algorithm $\mathcal{A} : L(B) \to \{\,\text{true}, \text{false}\,\}$

Output: LLL reduced basis $\mathsf{B}' = \hat{\mathsf{B}}'\mathsf{R}'$ of L such that
- $0.99\|\mathbf{b}'_1\|^2 < \min\{\, \|\mathbf{v}\|^2 \mid \mathbf{v} \in S_{u_{\max},\mathsf{B}'} \,\}$ or
- $\mathsf{CSSS}(\mathsf{B}', \mathsf{R}', u_{\max}) = \text{false}$.

```
procedure PoolSR(B, u_max, δ, s, A)
    (B, 𝔟, R) ← LLL(B, δ)                    /* B = B̂R, 𝔟 = (‖b̂_1‖², ..., ‖b̂_n‖²) */
    while CSSS(𝔟, u_max) = true do
        t ← 0                                /* Invariant: ‖p_1‖ ≤ ··· ≤ ‖p_t‖, t ≤ s */
        for x from 0 to 2^{u_max} − 1 do
            v ← Sample(B, R, x)
            if ‖v‖² ≤ 0.99‖b_1‖² or A(v) = true then
                j_v ← min{ j ∈ { 1, ..., t + 1 } | j = t + 1 or ‖v‖ ≤ ‖p_j‖ }
                [p_1, ..., p_{t+1}] ← [p_1, ..., p_{j_v − 1}, v, p_{j_v}, ..., p_t]
                t ← min{ s, t + 1 }
            end if
            if t > 0 and ‖p_1‖² ≤ 0.99‖b_1‖² then
                break
            else if x = 2^{u_max} − 1 then
                terminate("no short vector")
            end if
        end for
        (B, 𝔟, R) ← LLL([p_1, ..., p_t, b_1, ..., b_n], δ)
    end while
    return B
    terminate("further progress too unlikely")
end procedure
```

restriction is modeled by the acceptor $\mathcal{C}_\tau$ with threshold $\tau \geq 0.99$,

$$\mathcal{C}_\tau(\mathbf{v}) = (\|\mathbf{v}\|^2 \leq \tau\|\mathbf{b}_1\|^2)\,. \tag{3.28}$$

E. g., $\mathcal{C}_1$ ensures that the pool holds only vectors that are shorter than $\mathbf{b}_1$.

In general, we want to accept also vectors that are longer than $\mathbf{b}_1$ provided they are still significantly shorter than the longest base vector. However, the optimal choice of τ for a given basis is not obvious. This is addressed by the randomized acceptor $\mathcal{R}_{\tau,\alpha}$ that makes the selection of the pool vectors more fuzzy. It accepts vectors $\mathbf{v}$ such that $\|\mathbf{v}\|^2 > \tau\|\mathbf{b}_1\|^2$ as pool vector candidates with a chance depending on their

Algorithm 7 Randomized acceptor $\mathcal{R}_{\tau,\alpha}$

Input:
- vector $\mathbf{v} \in \mathbb{Z}^n$
- squared norm $\|\mathbf{b}_1\|^2$ of reference vector
- threshold $\tau \geq 0.99$
- fuzziness coefficient $\alpha > 0$
- uniform (pseudo) random number generator Rand on $[0, 1)$

Output: $\mathcal{R}_{\tau,\alpha} \in \{\text{true}, \text{false}\}$ such that
- $\mathcal{R}_{\tau,\alpha} = \text{true}$ if $\|\mathbf{v}\|^2 \leq \tau\|\mathbf{b}_1\|^2$,
- $\Pr[\mathcal{R}_{\tau,\alpha} = \text{true}] = e^{\alpha(\tau - \|\mathbf{v}\|^2/\|\mathbf{b}_1\|^2)}$ else.

```
procedure R_{τ,α}(v, ‖b_1‖², τ, α, Rand)
    r ← Rand()
    p ← e^{α(τ − ‖v‖²/‖b_1‖²)}
    return r ≤ p
end procedure
```

length. More precisely, for any vector $\mathbf{v} \in L(\mathsf{B})$,

$$\Pr[\mathcal{R}_{\tau,\alpha}(\mathbf{v}) = \text{true}] = \min\left\{1,\, e^{\alpha(\tau - \|\mathbf{v}\|^2/\|\mathbf{b}_1\|^2)}\right\}. \tag{3.29}$$

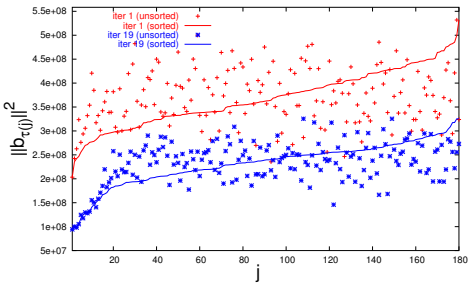
For instance, $\mathcal{R}_{1,\ln(2)}$ accepts any vector shorter than $\mathbf{b}_1$. If $\mathbf{v}$ is twice as long as $\mathbf{b}_1$ then $\mathcal{R}_{1,\ln(2)}$ outputs a uniformly distributed random bit. If $\mathbf{v}$ is three time as long as $\mathbf{b}_1$ then $\mathcal{R}_{1,\ln(2)}$ accepts $\mathbf{v}$ only in one out of four cases, and so on. Alg. 7 demonstrates that the implementation of $\mathcal{R}_{\tau,\alpha}$ is straightforward.

3.5.2. Empirical Behavior of PoolSR

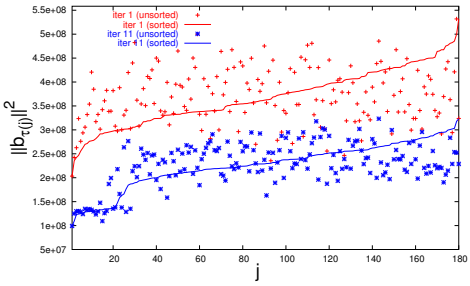
If used with the acceptor $\mathcal{R}_{\tau,\alpha}$, then PoolSR is not deterministic anymore. If we run PoolSR several times with the same input then we observe in fact differences in the runtime and the lengths of the base vectors. We discuss below the effects of the parameters using examples that exhibit typical behavior.

Fig. 3.5 on the facing page shows the squared lengths of the base vectors before and after PoolSR with $\mathcal{A}_{\mathsf{SSR}}$, $\mathcal{R}_{1,1}$, and $\mathcal{R}_{1,25}$, respectively. In the latter experiments the maximum pool size was $s = 60$. The input basis was generated in the same way as described in Sect. 3.1.2. In contrast to the SSR result depicted in Diag. (a) – where the length of the first base vectors grows rapidly – we observe in Diag. (b) and (c) a block of vectors not much longer than $\mathbf{b}_1$ at the front of the bases. The length of this block varies in most cases between 15 and 30. After this block the base vectors suddenly become longer; the squared length gap between the leading block and the shortest of the remaining base vectors is typically between 30 % and 50 % of the squared lengths of the first base vectors.

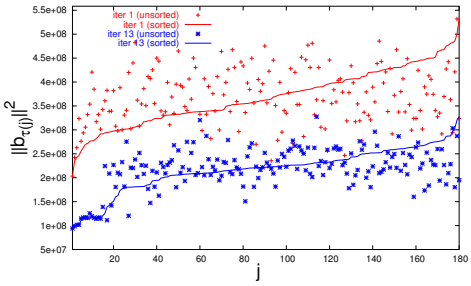
Compared with SSR, PoolSR does not significantly reduce the SVP approximation factor. With above input, the squared length of $\mathbf{b}_1$ in the PoolSR output basis ranged in several experiments from 8.2×10^7 to 9.9×10^7 whereas SSR achieves 9.4×10^7. We



(a) Poolsize $s = 1$, Acceptor $\mathcal{A}_{\mathsf{SSR}}$.



(b) Poolsize $s = 60$, Acceptor $\mathcal{R}_{1,1}$.



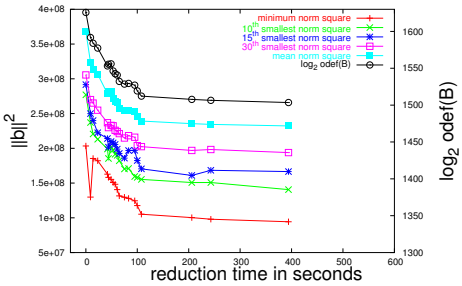
(c) Poolsize $s = 60$, Acceptor $\mathcal{R}_{1,25}$.

Figure 3.5.: Typical base vector length reduction by PoolSR.

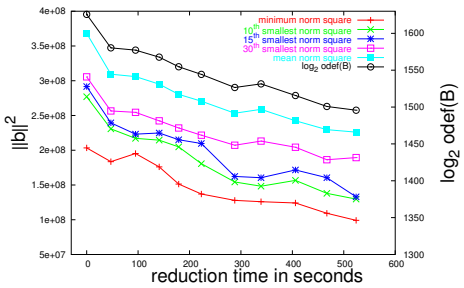
did not notice any influence of the acceptor parameter α either. The same holds for the reduction of the orthogonalization defect. The block of short vectors at the beginning of the basis is not large enough to significantly improve $\mathrm{odef}(B)$; the final value is in all three experiments $\approx 2^{1500}$.

The acceptor parameter α does affect the runtime behavior of PoolSR, though. With $\alpha = 1$, the pool grows almost always to its maximum size $s = 60$, even if a vector short enough to leave the inner loop is found after less than 2^{10} samples. The mean squared pool vector length varies from $1.2 \cdot \|\mathbf{b}_1\|^2$ to $1.6 \cdot \|\mathbf{b}_1\|^2$, the average factor is about 1.4. The maximum squared pool vector length is on average about $1.6 \cdot \|\mathbf{b}_1\|^2$. I. e., the bulk of the pool vectors is not much shorter than the shortest of the remaining base vectors. The BKZ reduction of the generating system formed by the pool and the former basis with so many vectors not reduced at all costs much time: In the experiment shown in Fig. 3.5(b), only 43 seconds were spent in the sampling loop, but 481 seconds in the BKZ reduction.

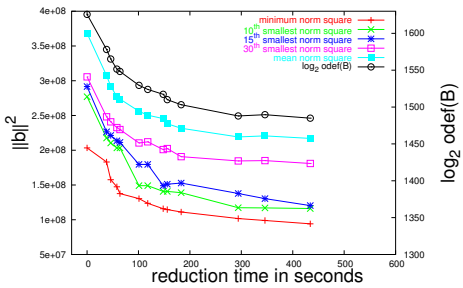
The situation is different with a large acceptor parameter $\alpha = 25$. The pool contains most of the time less than 5 vectors. On average, the squared norm of the longest pool vector does not exceed $1.1 \cdot \|\mathbf{b}_1\|^2$. In consequence, PoolSR spends much less time in the BKZ reduction of the generating system: In the experiment shown



(a) Poolsize $s = 1$, Acceptor $\mathcal{A}_{\mathsf{SSR}}$.



(b) Poolsize $s = 60$, Acceptor $\mathcal{R}_{1,1}$.



(c) Poolsize $s = 60$, Acceptor $\mathcal{R}_{1,25}$.

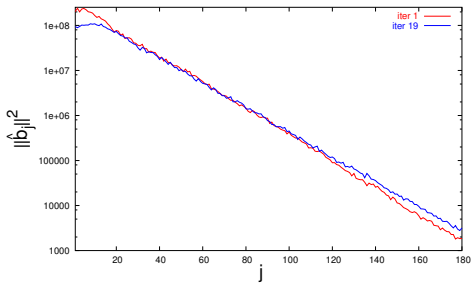
Figure 3.6.: Typical runtime behavior of PoolSR.

in Fig. 3.5(c), 285 seconds were spent on sampling, but only 149 seconds on BKZ reduction.

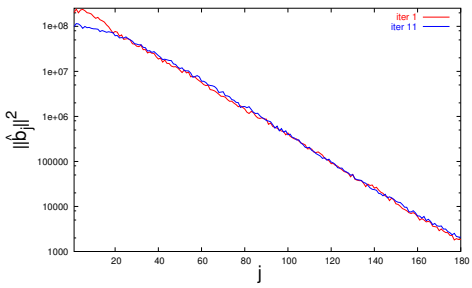
Fig. 3.6 exhibits these differences in the runtime behavior. In Diag. (b), each iteration of PoolSR with $\alpha = 1$ took approximately 40 to 60 seconds, no matter how many samples were required. The squared norm of the shortest base vector decreased at a moderate pace only. (It increased in the second iteration because $\mathbf{b}_1$ was not the shortest base vector.)

In contrast, we see in Diag. (c) that PoolSR with $\alpha = 25$ allows a much more rapid improvement of the shortest base vector; e. g., after less than 200 seconds, the minimum norm square was already down to $1.1{\times}10^8$. In fact, this behavior is quite similar to the behavior of SSR if applied to the same input basis: SSR terminated after 394 seconds of which the last three iterations accounted for 286 seconds. But the 15th smallest norm square in the output of SSR is about $1.7{\times}10^8$ whereas it is only about $1.2{\times}10^8$ in the PoolSR output.

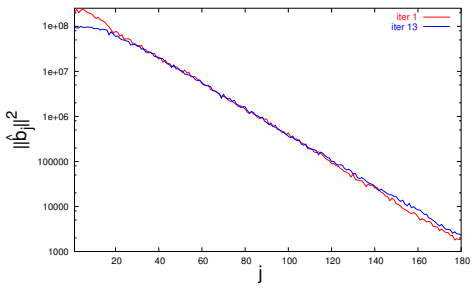
Like SSR, PoolSR significantly affects the first Gram-Schmidt vectors of the basis only. The lengths of $\hat{\mathbf{b}}_j$, $j > 30$, do not differ much in the input and the output basis. This observation explains why PoolSR does not generate a better SVP approximation factor than SSR: Since the effects of BKZ do not reach further into the basis if we



(a) Poolsize $s = 1$, Acceptor $\mathcal{A}_{\mathsf{SSR}}$.



(b) Poolsize $s = 60$, Acceptor $\mathcal{R}_{1,1}$.



(c) Poolsize $s = 60$, Acceptor $\mathcal{R}_{1,25}$.

Figure 3.7.: Typical Gram-Schmidt vector length after PoolSR.

prepare a generating system from the original basis and a larger pool compared to a generating system from the basis preceded by a single short vector, the success probability of the sampling loop in PoolSR is similar to that in SSR.

3.6. Short Projection Sampling Reduction

We describe in Sect. 3.6.1 the *Short Projection Sampling Reduction* algorithm (ShortProjectionSR) that aims at shaping the sequence $(\|\hat{\mathbf{b}}_j\|)_{j=1,\ldots,n}$ such that subsequent sampling iterations stand a greater chance to find a vector shorter than $\mathbf{b}_1$. We report typical empirical results with ShortProjectionSR in Sect. 3.6.2.

3.6.1. The ShortProjectionSR Algorithm

When we study the sampling reduction of NTRU lattices in Section 6.2, then we will encounter bases where $\mathbf{b}_1$ is orders of magnitude shorter than the longest Gram-Schmidt vector. It is therefore very unlikely for any feasible search space size $u_{\max}$ that Sample generates a vector shorter than $\mathbf{b}_1$. Even if we reduce a basis that approximates (GSA) reasonably well, the effects of the intermittent LLL-type reduction

Algorithm 8 ShortProjectionSR

Input:
- basis $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d \times n}$ of lattice L
- search space bound $u_{\max} \in \mathbb{N}$
- reduction factor $\gamma \in (0, 1)$
- set T of target indices, $\emptyset \neq T \subseteq \{ 1, \ldots, n - u_{\max} \}$
- LLL parameter $\delta \in (1/4, 1)$

Output: LLL reduced basis $\mathsf{B}' = \hat{\mathsf{B}}'\mathsf{R}'$ of L such that $\forall j \in T$:
- $\gamma \|\mathbf{b}'_j\|^2 < \min\{ \pi_j(\|\mathbf{v}\|^2) \mid \mathbf{v} \in S_{u_{\max}, \mathsf{B}'} \}$ or
- $\mathsf{CSSS}((\|\hat{\mathbf{b}}'_j\|^2, \ldots, \|\hat{\mathbf{b}}'_n\|^2), u_{\max}) = \text{false}$.

```
procedure ShortProjectionSR(B, u_max, γ, T, δ)
    (B, 𝔟, R) ← LLL(B, δ)                    /* B = B̂R, 𝔟 = (‖b̂_1‖², ..., ‖b̂_n‖²) */
    while CSSS'(𝔟, γ, T, u_max) = true do
        for x from 0 to 2^{u_max} − 1 do
            v ← Sample(B, R, x)
            t ← min{ j ∈ T | ‖π_j(v)‖² ≤ γ‖b̂_j‖² } ∪ { ∞ }
            if t ∈ T then
                break
            else if x = 2^{u_max} − 1 then
                terminate("no short vector")
            end if
        end for
        (B, 𝔟, R) ← LLL([b_1, ..., b_{t−1}, v, b_t, ..., b_n], δ)
    end while
    terminate("further progress too unlikely")
end procedure
```

on the length of the Gram-Schmidt vectors $\hat{\mathbf{b}}_j$ peters quickly out if $j \geq 30$, as seen in Fig. 3.2 and 3.7. This brings the Sampling Reduction soon to a halt. ShortProjection-SR addresses both problems by inserting sampled vectors in between the base vectors rather than to prepend them.

Schnorr remarks in [Sch03] that the very first base vectors often show a bad (GSA) approximation. He therefore suggests EShort, a modification of the sampling loop that searches a vector $\mathbf{v} \in S_{u,\mathsf{B}}$ such that $\|\pi_j(\mathbf{v})\|^2 \leq 0.99\|\hat{\mathbf{b}}_j\|^2$ for some $j \leq 10$. The new generating system of $L(\mathsf{B})$ is then formed by inserting $\mathbf{v}$ in front of $\mathbf{b}_j$. Therefore, Sampling Reduction with EShort may proceed longer than RSR because more sample vectors can trigger another iteration of the outer loop and the LLL-type reduction may affect the length of some more Gram-Schmidt vectors. However, EShort sticks to modifying the generating system near the front of the basis.

ShortProjectionSR advances on EShort by shifting the focus from making the outer Sampling Reduction loop proceed some more iterations to changing the lengths of

the Gram-Schmidt vectors in the middle of the basis. We want to exploit two effects: If we insert vectors with short projections then we make the hump smaller that we observed in Fig. 3.2. Therefore, the likelihood to sample a vector shorter than $\mathbf{b}_1$ in further Sampling Reduction iterations grows. Second, recall that all LLL-type reductions ensure that $\|\pi_{j-1}(\mathbf{b}_j)\|^2/\|\pi_{j-1}(\mathbf{b}_{j-1})\|^2$, $j \in \{2, \ldots, n\}$, does not fall below some threshold. Vectors with short projections inserted in the middle of the generating system therefore cause the LLL-type reduction to modify the generating system where we did not observe much effect with SSR. In particular, we hope to see a decrease in the lower bound

$$\alpha = \frac{\min\{\, \|\mathbf{b}_j\| \mid j = 1, \ldots, n \,\}}{\min\{\, \|\hat{\mathbf{b}}_j\| \mid j = 1, \ldots, n \,\}}$$

on the approximation factor we obtained. We introduce an explicit parameter γ that specifies the factor by which the inserted vector's projection must be shorter than the respective $\hat{\mathbf{b}}_j$. The smaller γ the more likely it is that the inserted vector forces the LLL-type reduction to modify the basis around the insertion index.

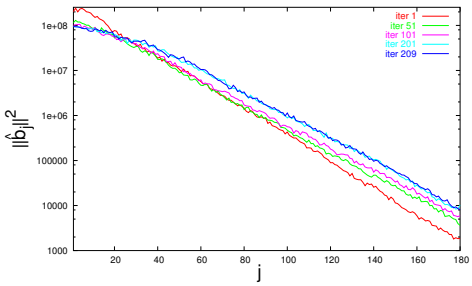
The ShortProjectionSR algorithm is presented in Alg. 8. It takes – compared with SSR– two additional parameters: The reduction factor γ and the set T of target indices. T specifies at which positions a sample vector may be inserted into the lattice basis to form the new generating system. The reduction factor determines when a sample vector projection is considered short enough; ShortProjectionSR leaves the sampling loop if there is an index $j \in T$ such that the sample vector $\mathbf{v}$ satisfies $\|\pi_j(\mathbf{v})\|^2 \leq \gamma\|\hat{\mathbf{b}}_j\|$. The minimum index for which this condition holds is stored in the variable t. Then $\mathbf{v}$ is inserted in B at column t and this generating system is reduced again with an LLL-type reduction as in SSR.

The various Check Search Space Size algorithms discussed above assumed that we search a vector $\mathbf{v}$ subject to $\|\mathbf{v}\|^2 \leq 0.99\|\mathbf{b}_1\|^2$ whence we need to modify them as well. Note that $\mathbf{b}_1 = \hat{\mathbf{b}}_1$ and $\pi_1(\mathbf{v}) = \mathbf{v}$. So $\mathsf{CSSS}((\|\hat{\mathbf{b}}_j\|^2, \ldots, \|\hat{\mathbf{b}}_n\|^2), u_{\max})$ actually returns true if and only if it estimates the probability to find a vector $\mathbf{v} \in S_{u_{\max},\mathsf{B}}$ such that $\|\pi_j(\mathbf{v})\|^2 \leq 0.99\|\hat{\mathbf{b}}_j\|^2$ as sufficient. In all considered CSSS implementations it is straight forward to replace the factor 0.99 by γ. T may contain more than one potential insertion point. We can easily account for that by computing the logical *or* of $\mathsf{CSSS}((\|\hat{\mathbf{b}}_j\|^2, \ldots, \|\hat{\mathbf{b}}_n\|^2), u_{\max})$ for all $j \in T$. We name the CSSS implementations that were adapted in such a manner CSSS′.

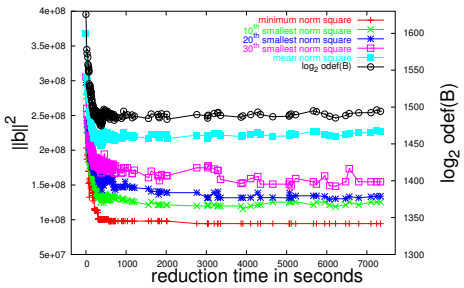
If we specialize $\gamma = 0.99$ and $T = \{\, 1 \,\}$, then ShortProjectionSR reduces to SSR; so ShortProjectionSR is in fact a generalization. Furthermore, if we apply ShortProjectionSR with $\gamma = 0.99$ and $T = \{\, 1, \ldots, 10 \,\}$, then we obtain SSR combined with Schnorr's EShort.

3.6.2. Empirical Behavior of ShortProjectionSR

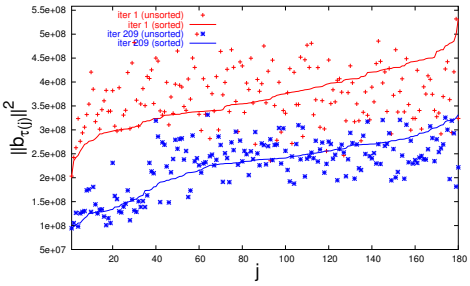
We first describe how the combination of SSR with Schnorr's EShort – in the following named EShort-ShortProjectionSR – performs. We then consider the effects of ShortProjectionSR if used to overcome the lack of change in $\|\hat{\mathbf{b}}_j\|^2$ for $j > 30$. In both cases



(a) Gram-Schmidt vector length.



(b) Runtime behavior.



(c) Base vector length.

Figure 3.8.: Typical effect of ShortProjectionSR with $\gamma = 0.99$ and $T = \{1, \ldots, 10\}$.

the example lattice used is the same as in Sect. 3.5.2 and the LLL-type reduction is always BKZ with $(\delta, \beta) = (0.99, 5)$.

Comparing Fig. 3.8 with Fig. 3.5 and Fig. 3.6, two observations are most striking: First, neither the minimum nor the mean base vector norm square nor the orthogonalization defect differ significantly between the output of SSR and EShortShortProjectionSR. However, the base vector length does not increase as rapidly in the latter as in the former. Similar to PoolSR with $\mathcal{R}_{1,25}$, there are base vectors not much longer than $\mathbf{b}_1$. These vectors do not form such a distinct plateau, though. The lengths of the first base vectors are more scattered.

Second, ShortProjectionSR quickly reduced the minimum and average base vector length, like SSR; this decrease came to a stop after about 150 iterations (about 500 seconds). I. e., each iteration took only few seconds, most of which were spent in the BKZ reduction. (A more optimized implementation that imposes less overhead for each iteration is likely to speed up this phase of the reduction.) Fig. 3.8(b) shows that at this point the distribution of the base vector lengths is similar to the distribution after PoolSR with $\mathcal{R}_{1,25}$ (Fig. 3.6(c)).

After that, ShortProjectionSR kept finding suitable vectors – sampling up to 2^{22} elements – for further 60 iterations. On average, 2^{16} to 2^{17} samples were required before

a vector was found that could be inserted in the generating system, accumulating to somewhat more than 7000 seconds runtime. (The runtime of the intermittent BKZ reductions was negligible in comparison.) But these more expensive iterations did not improve $\|\mathbf{b}_1\|^2$ anymore. There was only a minimal improvement in the length of the 20th and 30th shortest base vectors. The mean squared length vacillated around 10 % above its minimum value.

This leaves the impression that, with respect to the base vector length, EShort-ShortProjectionSR performs similar to PoolSR if terminated as soon as there is no more progress. However, we made a curious observation that causes us to prefer PoolSR over EShort-ShortProjectionSR: We repeated some of our experiments on a notebook with an Intel Pentium 4 Mobile CPU at 1.7 GHz and 1 GByte RAM. In general, the notebook took about 1.25 as long as the PC to obtain the same results. But the intermittent BKZ reduction in the ShortProjectionSR iterations took about 3 times as long as on the PC! (The performance of the sampling loop was not notably slower than expected, though.) Thus, on the notebook, EShort-ShortProjectionSR required about 1500 seconds to compute a result comparable to PoolSR with $\mathcal{R}_{1,25}$, given the same input as above. PoolSR did not suffer the same slow-down in its BKZ reduction, it took typically between 600 and 700 seconds to complete. We can only speculate about the cause of this phenomenon: The implementations of the reduction algorithms used were always the same. But EShort-ShortProjectionSR has a more global effect on the Gram-Schmidt decomposition of the bases than PoolSR, cf. Fig. 3.8(a) versus Fig. 3.7. This seems to trigger a code flow and memory access pattern that the notebook hardware cannot handle well due to, e. g., a different RAM interface or less first level cache. The ShortProjectionSR is more likely to be affected by such hardware differences since it stresses the BKZ reduction more than PoolSR. So, even though our experiments on our primary test hardware indicated a similar performance of EShort-ShortProjectionSR and PoolSR, we prefer the latter to compute short basis vectors.

The more global effect of EShort-ShortProjectionSR on the Gram-Schmidt decomposition can also be an advantage, though: If we are interested in minimizing the guaranteed approximation factor α, then this algorithm is more effective. We can deduce from the PoolSR result shown in Fig. 3.7 that the approximation factor is at most 184, whereas the EShort-ShortProjectionSR yields $\alpha \leq 109$. If we apply ShortProjectionSR with $\gamma = 0.99$ and $T = \{1, \ldots, 30\}$, then we obtain in 4000 seconds the even better approximation factor bound $\alpha \leq 80$ but the worse minimum squared base vector length $\|\mathbf{b}_1\|^2 \approx 1.2 \times 10^8$.

There is no reason to restrict the target indices set T to consecutive numbers $\{1, \ldots, t\}$ for some t. For example, when we study the effects of sampling reduction on the reduction of NTRU lattice bases, ShortProjectionSR with $T = \{t\}$ for some $t > 1$ will assume the role of SSR because the Gram-Schmidt vectors $\hat{\mathbf{b}}_j$ with $j < t$ are orders of magnitude shorter than $\hat{\mathbf{b}}_t$; in that situation, any attempt to find a vector shorter than $\mathbf{b}_1$ in a reasonably sized search space is doomed to failure.

When reducing bases that originally met (GSA) the freedom in choosing target

stage	algorithm	parameters	$u_{\max}$	CSSS′	t_{stage}	N_{stage}
1	PoolSR	$\gamma = 0.99, s = 60, \mathcal{A} = \mathcal{R}_{1,25}$	25	$\text{CSSS}'_{\text{event}}$	500	75
2	ShortProjectionSR	$\gamma = 0.75, T = \{1, 21, \ldots, 25\}$	24	$\text{CSSS}'_{\text{event}}$	150	60
3	ShortProjectionSR	$\gamma = 0.75, T = \{1, 41, \ldots, 45\}$	24	$\text{CSSS}'_{\text{event}}$	150	60
4	ShortProjectionSR	$\gamma = 0.75, T = \{1, 61, \ldots, 65\}$	24	$\text{CSSS}'_{\text{event}}$	150	60
5	PoolSR	$\gamma = 0.99, s = 60, \mathcal{A} = \mathcal{R}_{1,25}$	25	$\text{CSSS}'_{\text{event}}$	500	75
6	ShortProjectionSR	$\gamma = 0.85, T = \{1, 21, \ldots, 25\}$	24	$\text{CSSS}'_{\text{event}}$	150	60
7	ShortProjectionSR	$\gamma = 0.85, T = \{1, 41, \ldots, 45\}$	24	$\text{CSSS}'_{\text{event}}$	150	60
8	ShortProjectionSR	$\gamma = 0.85, T = \{1, 61, \ldots, 65\}$	24	$\text{CSSS}'_{\text{event}}$	150	60
9	ShortProjectionSR	$\gamma = 0.99, T = \{1, \ldots, 5\}$	24	$\text{CSSS}'_{\text{event}}$	150	60

Table 3.1.: Algorithms and parameters used in Fig. 3.9

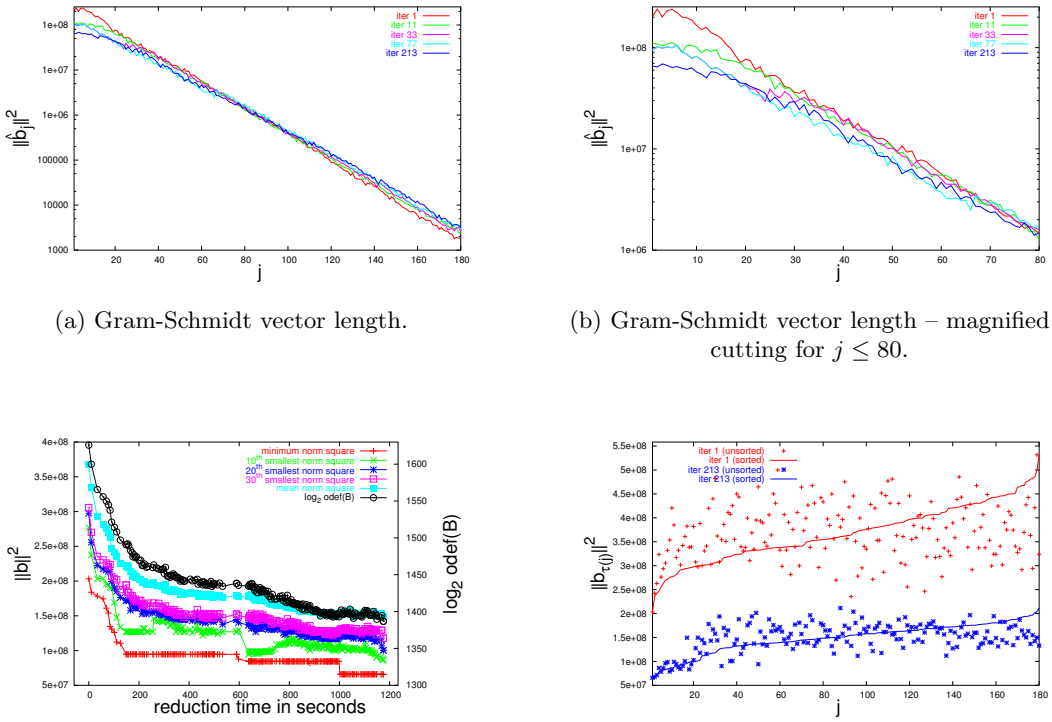
indices can be helpful as well. If we apply PoolSR to such a basis and subsequently run ShortProjectionSR with T chosen such that specific Gram-Schmidt vectors become shorter, then, in consequence, further iterations of PoolSR are more likely to succeed.

Fig. 3.9 exhibits this effect. In this experiment we reduced the input basis in nine stages; the algorithms and parameters used in each stage are given in Table 3.1. We moved on to the next stage whenever CSSS′ returned false, more than t_{stage} seconds had already passed since the beginning of the current stage, or the reduction algorithm's outer loop had been already iterated N_{stage} times, whatever happened first. The intermittent LLL-type reduction was always BKZ with $(\delta, \beta) = (0.99, 5)$.

The second, third, and fourth graph in Fig. 3.9(a) and in its magnified cutting Fig. 3.9(b) show the squared length of the Gram-Schmidt vectors at the end of stage 1, 2, and 4, respectively. We see that the initial PoolSR did not significantly change the length of the Gram-Schmidt vectors $\hat{\mathbf{b}}_j$ with $j \geq 20$. The following ShortProjectionSR decreased $\|\hat{\mathbf{b}}_j\|^2$ by up to 20 % for $j \in \{10, \ldots, 30\}$. As before, the squared length of the Gram-Schmidt vectors did not significantly change for $j > 30$. At the end of stage 4, $\|\hat{\mathbf{b}}_j\|^2$ was reduced by 10 % to 20 % for $10 \leq j \leq 60$.

The first stage had ended after 11 iterations in 149 seconds since $\text{CSSS}'_{\text{event}}$ estimated that the search space size had to be at least $u = 30$ for further progress to be likely. At the beginning of stage 5, the reduction of $\|\hat{\mathbf{b}}_j\|^2$ for $j \leq 60$ had caused this estimate to drop to $u = 19$. In fact, a sufficiently short vector was found in the search space after enumeration of only $2^{12.64}$ vectors. This demonstrates that we can use ShortProjectionSR to boost PoolSR's success probability.

In the end, $\|\mathbf{b}_1\|^2$ was reduced to 6.6×10^7, i. e., to a third of its original length. Even more notably, the combination of PoolSR and ShortProjectionSR improved the basis globally such that the average squared base vector length was finally only 1.5×10^8, a reduction by 25 % to 40 % compared with the results of the previous algorithms. We can thus achieve approximately the same length reduction for all base vectors, not only for the very first ones as with SSR. The orthogonalization defect was also improved to 2^{1387}, the minimum value for all Sampling Reductions of this particular



(a) Gram-Schmidt vector length.

(b) Gram-Schmidt vector length – magnified cutting for $j \leq 80$.

(c) Runtime behavior.

(d) Base vector length.

Figure 3.9.: Typical effect of combined PoolSR and ShortProjectionSR.

basis. We pay for this improvement with a roughly doubled runtime, compared with PoolSR.

The parameters given in Table 3.1 are often a good starting point to obtain acceptable results with ShortProjectionSR. In general, though, the user has to make an intelligent choice of T, based on the length of the intermediate Gram-Schmidt vectors. If T is too large or is not changed often enough, then ShortProjectionSR typically starts to increase the length of $\hat{\mathbf{b}}_j$ for large j as shown, e. g., in Fig. 3.8(a). That causes the success probability of further PoolSR iterations to drop rapidly. Therefore, ShortProjectionSR is best suited for an interactive setting.

3.7. Further Generalizations

The target indices set T in ShortProjectionSR enables us to apply Sampling Reduction to bases that grossly violate (GSA) due to a peek of the Gram-Schmidt vector lengths in the front half of the basis, effectively ignoring all vectors before that peek. We describe a generalization of the search space that facilitates something similar if

there is a jump in the length of the trailing Gram-Schmidt vectors. We conclude this chapter with two remarks on distributed implementations of Sampling Reduction and on the reduction of bases that are only implicitly given by Gram matrices.

3.7.1. Search Space Generalization

The definition of the search space $S_{u,\mathsf{B}}$ in 2.3 was motivated by the empiric observation that the larger j the smaller is $\|\hat{\mathbf{b}}_j\|^2$. That is not necessarily so, even for LLL-reduced bases. (Recall, LLL defines a lower bound only for the ratio $\|\hat{\mathbf{b}}_{j-1}\|^2/\|\hat{\mathbf{b}}_j\|^2$.) But if $\hat{\mathbf{b}}_n$ is already large, than it is unlikely that $S_{u,\mathsf{B}}$ contains any short vector $\mathbf{v}$ because $\|\mathbf{v}\|^2 \geq \|\hat{\mathbf{b}}_n\|^2$. In such a case we would like to ignore $\hat{\mathbf{b}}_n$ and look for a short vector in the sublattice generated by $[\mathbf{b}_1, \ldots, \mathbf{b}_{n-1}]$.

Similarly, let $J \subset \{1, \ldots, n\}$ such that $G = \{\hat{\mathbf{b}}_j \mid j \in J\}$ is the set of the $u+1$ shortest Gram-Schmidt vectors of the basis B. The Sample algorithm generates vectors $\mathbf{v} = \sum_j \nu_j \hat{\mathbf{b}}_j$ such that $\nu_j^2 > 1/4$ only if $n-u \leq j$. Assume $\{n-u, \ldots, n-1\}$ is not contained in J. Then Sample is suboptimal because the expected length of the vectors it computes is larger than the length we could expect if $\nu_j^2 > 1/4$ implied $j \in J$.

Third, recall that $E[\,\|\mathbf{v}\|^2 \mid \mathbf{v} \in S_{u,\mathsf{B}}\,] = \sum_j \frac{1}{3} r_j^2 \|\hat{\mathbf{b}}_j\|^2$ by (RA) where $\nu_j \in (-r_j, r_j]$. Therefore, if $\|\hat{\mathbf{b}}_k\|^2 < \|\hat{\mathbf{b}}_l\|^2$ and we can arrange a search space $S'_{u,\mathsf{B}}$ such that

$$r'_j = \begin{cases} 2r_k & \text{if } j = k, \\ r_l/2 & \text{if } j = l, \text{ and} \\ r_j & \text{else,} \end{cases}$$

then $E[\,\|\mathbf{v}'\|^2 \mid \mathbf{v}' \in S'_{u,\mathsf{B}}\,] < E[\,\|\mathbf{v}\|^2 \mid \mathbf{v} \in S_{u,\mathsf{B}}\,]$.

The generalized search space $V_{\mathbf{c},\mathsf{B}}$ defined below enables us to exploit all these observations. The Sample and CSSS implementations can be easily adapted to work with $V_{\mathbf{c},\mathsf{B}}$.

Definition 16. *Let $\mathsf{B} \in \mathbb{Z}^{d\times n}$ be a lattice basis with Gram-Schmidt decomposition $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$. Let $\mathbf{c} = (c_1, \ldots, c_n) \in \mathbb{N}^n$ and assume there is $j_0 \in \{1, \ldots, n\}$ such that $c_j = 0$ if and only if $j > j_0$. Then the generalized search space $V_{\mathbf{c},\mathsf{B}}$ is the set of all lattice vectors $\mathbf{v} = \sum_{j=1}^n \nu_j \hat{\mathbf{b}}_j \in L(\mathsf{B})$ subject to*

$$\nu_j \in \begin{cases} (-c_j/2, c_j/2] & \text{if } 1 \leq j < j_0, \\ \{1, \ldots, c_j\} & \text{if } j = j_0, \\ \{0\} & \text{if } j_0 < j \leq n, \end{cases} \tag{3.30}$$

for all $j \in \{1, \ldots, n\}$.

The algorithm GenSample presented in Alg. 9 on the next page is the obvious adaption of Sample that generates vectors from $V_{\mathbf{c},\mathsf{B}}$.

Algorithm 9 GenSample

Input:
- unit upper triangular matrix $\mathsf{R} = [\mathbf{r}_1, \ldots, \mathbf{r}_n] \in \mathbb{Q}^{n \times n}$,
- lattice basis $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d \times n}$ with Gram-Schmidt decomposition $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$,
- $\mathbf{c} \in \mathbb{N}^n$ as in Def. 16,
- $x \in \mathbb{N}$

Output: $\mathbf{v} \in V_{\mathbf{c},\mathsf{B}}$

```
procedure Sample (B, R, c, x)
    for j from n downto 1 do
        if c_j > 0 then
            (x, z) ← (⌊x/c_j⌋, x mod c_j)
            if j = n or c_{j+1} = 0 then
                (v, ν) ← (x + 1)(b_j, r_j)    /* ν = (ν_1, ..., ν_n)^t,  ν_{j+1} = ··· = ν_n = 0 */
            else
                z ← (−1)^z ⌊z/2⌋                          /* z ∈ Z ∩ (−c_j/2, c_j/2] */
                y ← ⌈ν_j − 1/2⌉                           /* ν_j − y ∈ (−1/2, 1/2] */
                if ν_j − y ≤ 0 then
                    y ← y − z                             /* ν_j − y ∈ (−c_j/2, c_j/2] */
                else
                    y ← y + z                             /* ν_j − y ∈ (−c_j/2, c_j/2] */
                end if
                (v, ν) ← (v, ν) − y(b_j, r_j)             /* ν_j ← ν_j − y */
            end if
        end if
    end for
    return v
end procedure
```

Proposition 24. *Let* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R} \in \mathbb{Z}^{d \times n}$, $\mathbf{c} = (c_1, \ldots, c_n) \in \mathbb{N}^n$, *and* $j_0 \in \{1, \ldots, n\}$ *as in Def. 16. Set* $N = \prod_{j=1}^{j_0} c_j \geq 1$. *Then*

$$f : \{0, \ldots, N-1\} \to V_{\mathbf{c},\mathsf{B}} : x \mapsto \mathsf{GenSample}(\mathsf{B}, \mathsf{R}, \mathbf{c}, x)$$

is a bijection. In particular, $|V_{\mathbf{c},\mathsf{B}}| = N$.

Proof. Note that $\mathbf{v}$ and $\boldsymbol{\nu}$ will always be eventually initialized in the first branch of the innermost **if** block because $c_1 > 0$ by the precondition on $\mathbf{c}$. In the second branch, z is assigned an integer such that

$$z \in \begin{cases} \{-(c_j - 2)/2, \ldots, c_j/2\} & \text{if } c_j \text{ is even,} \\ \{-(c_j - 1)/2, \ldots, (c-1)/2\} & \text{if } c_j \text{ is odd.} \end{cases}$$

More precisely, z is $0, 1, -1, 2, -2, \ldots$ if $x \bmod c_j$ is $0, 1, 2, 3, 4, \ldots$, respectively. Then z is used to choose one of the c_j possible values for ν_j in the interval $(-c_j/2, c_j/2]$.

Otherwise, the proof is analogous to the proofs of Prop. 7 and 8. □

GenSample constitutes the innermost loop of all Sampling Reduction variants. In practice, we therefore restrict all nonzero c_j to powers of 2 so computing $x \bmod c_j$ reduces to extracting the least significant bits of x and computing $\lfloor x/c_j \rfloor$ is simply a bit shift.

We immediately see that $V_{\mathbf{c},\mathsf{B}} = S_{u,\mathsf{B}}$ if $j_0 = n$ and

$$c_j = \begin{cases} 2 & \text{if } n - u \leq j < n, \\ 1 & \text{else.} \end{cases}$$

If the last $n - j_0$ Gram-Schmidt vectors are particularly long, then $V_{\mathbf{c}',\mathsf{B}}$ with

$$c'_j = \begin{cases} 0 & \text{if } n - j_0 < j, \\ 2 & \text{if } n - u - j_0 \leq j < n - j_0, \\ 1 & \text{else.} \end{cases}$$

is likely to contain shorter vectors than $S_{u,\mathsf{B}}$. We can address the second issue mentioned above by assigning c'_j a value greater than 1 if and only if $j \in J$. Finally, we can try to exploit third opportunity to optimize the expected squared length of the sample vectors pointed out above by choosing $c_j > 2$ for those c_j corresponding to the shortest Gram-Schmidt vectors.

$\mathsf{CSSS}_{\mathsf{triv}}$ can obviously be used in combination with GenSample without modification. For $\mathsf{CSSS}_{\mathsf{event}}$ we have to adapt ExpLength so it takes the following, generalized version of equation (3.2) into account:

$$s_j = \begin{cases} \frac{1}{12} c_j r^{k-j} & \text{if } 1 \leq j < k, \\ \frac{1}{12} c_j^2 & \text{if } k \leq j < j_0, \\ \frac{1}{2}(c_j + 1) & \text{if } j = j_0, \\ 0 & \text{else.} \end{cases} \tag{3.31}$$

Furthermore, we have to replace the permutation $\sigma_{u,\mathsf{B}}$ by $\tau_{\mathbf{c},\mathsf{B}}$ such that

$$\begin{aligned} &\tau_{\mathbf{c},\mathsf{B}}(j) = j && \text{if } j \geq j_0, \\ &\tau_{\mathbf{c},\mathsf{B}}(i) < \tau_{\mathbf{c},\mathsf{B}}(j) && \text{if } i, j \in \{1, \ldots, j_0 - 1\} \text{ and } c_i < c_j, \end{aligned} \tag{3.32}$$

and

$$\|\hat{\mathbf{b}}_{\tau_{\mathbf{c},\mathsf{B}}(i)}\| \geq \|\hat{\mathbf{b}}_{\tau_{\mathbf{c},\mathsf{B}}(j)}\| \quad \text{if } i < j < j_0 \text{ and } c_i = c_j. \tag{3.33}$$

The adaption of $\mathsf{CSSS}_{\mathsf{Fourier}}$ is even simpler: We compute

$$(t_1, \ldots, t_{j_0-1}) \leftarrow (c_1 \|\hat{\mathbf{b}}_1\|, \ldots, c_{j_0-1} \|\hat{\mathbf{b}}_{j_0-1}\|)/(2\|\mathbf{b}_1\|)\,.$$

The expected contribution of $\hat{\mathbf{b}}_{j_0}$ to the squared length of the sample vectors is

$$E := E[\,\nu_{j_0}^2 \|\hat{\mathbf{b}}_{j_0}\|^2 \mid \textstyle\sum_{j=0}^{n} \nu_j \hat{\mathbf{b}}_j \in V_{\mathbf{c},\mathsf{B}}\,] = \textstyle\sum_{l=1}^{c_{j_0}} l^2 \|\hat{\mathbf{b}}_{j_0}\|^2 / c_{j_0}$$
$$= (c_{j_0}+1)(2c_{j_0}+1)\|\hat{\mathbf{b}}_{j_0}\|^2/6\,.$$

Then the success probability is $\mathsf{Ratio}(\gamma - E/\|\mathbf{b}_1\|^2, (t_1, \ldots, t_{j_0-1}))$.

3.7.2. Distributed Sampling

The LLL algorithms manipulates the basis in each iteration. This implies that previous attempts to parallelize lattice basis reduction [Hec95, Wet98] had to rely on a very tight communication network – preferably memory shared by all computing nodes – and that the algorithm had to be broken down in very fine granular tasks that were assigned to the available nodes, resulting in a lot of overhead for the management of these tasks. Approaches that speculatively computed the Gram-Schmidt coefficients in columns $l \in \{\, j, \ldots, n \,\}$ while the main LLL algorithm was still at stage j had to discard all results every time LLL swapped two base vectors and therefore gained a moderate advantage only.

Sampling Reduction is different since it does not modify the basis during the sampling phase at all. It runs the same algorithm GenSample again and again for only slightly different input data. This is an algorithm pattern for which SIMD-parallelization (single instruction, multiple data) excels. Blochinger, Küchlin, Ludwig, and Weber describe in [BKLW99] a software package DOTS that supports the distribution of search problems as the one encountered in Sampling Reduction within loose networks of heterogenous machines, say the workstations in a department LAN.

To implement such a distributed Sampling Reduction we partition the search space in disjunct parts $V_{\mathbf{c},\mathsf{B}} = \bigcup_{j=1}^{N} V_j$ where N is typically a small multiple of the number of available nodes. Then we schedule the search space parts on the computing nodes; for this we need to transfer only once the basis and the specification of the V_j assigned. As soon as a node has exhausted its section of the search space, the next available V_j is scheduled on this node. This continues until either a node returns a sufficiently short vector – in which case all other nodes are notified and abort their task – or all V_j were searched without success.

If N is large enough to keep all nodes busy but the search space parts are not too small to avoid unnecessary communication overhead between the nodes and the scheduler, then distributed systems like DOTS have an efficiency close to 100 %, i. e., the attainable speedup scales linearly in the number of nodes. For instance, our primary test hardware computes 2^{16} samples in dimension 180 in about 10 to 15 seconds. So if each search packet covers at least 2^{16} vectors, then we expect that a distributed implementation results in a very high efficiency.

3.7.3. Gram Matrices

In some applications the lattices are only implicitly given by Gram matrices.

Definition 17. *A matrix* $\mathsf{G} \in \mathbb{R}^{n\times n}$ *is a* Gram matrix *if and only if* G *is symmetric positive semi-definite, i. e.,*

$$\mathsf{G} = \mathsf{G}^t \qquad \textit{and} \qquad \mathbf{x}^t\mathsf{G}\mathbf{x} \geq 0 \quad \textit{for all } \mathbf{x} \in \mathbb{R}^n.$$

For any lattice basis $\mathsf{B} \in \mathbb{Z}^{d\times n}$, the corresponding Gram matrix is $\mathsf{G} = \mathsf{B}^t\mathsf{B} \in \mathbb{Z}^{n\times n}$. Conversely, a non-singular Gram matrix $\mathsf{G} \in \mathbb{Z}^{n\times n}$ determines the corresponding lattice basis (and thereby the lattice) up to an isometry, i. e., there is a lattice basis $\mathsf{B} \in \mathbb{R}^{n\times n}$ such that $\mathsf{G} = \mathsf{B}^t\mathsf{B}$, and if $\mathsf{B}' \in \mathbb{R}^{n\times n}$ with $\mathsf{G} = \mathsf{B}'^t\mathsf{B}'$, then there is a matrix $\mathsf{O} \in \mathbb{R}^{n\times n}$ such that $\mathsf{O}^t\mathsf{O} = \mathsf{I}_n$ is the identity matrix and $\mathsf{B}' = \mathsf{O}\mathsf{B}$.

Note that the Gram-Schmidt decompositions of two lattice bases corresponding to the same Gram matrix differ only in the direction of the Gram-Schmidt vectors, i. e., if $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$ and $\mathsf{B}' = \hat{\mathsf{B}}'\mathsf{R}'$, then $\mathsf{R}' = \mathsf{R}$ and $\hat{\mathsf{B}}' = \mathsf{O}\hat{\mathsf{B}}$ with $\mathsf{O}^t\mathsf{O} = \mathsf{I}_n$. In particular, $\|\hat{\mathbf{b}}'_j\|^2 = \|\hat{\mathbf{b}}_j\|^2$ for all $j \in \{1, \ldots, n\}$. Since LLL-type reductions require only the squared lengths of the Gram-Schmidt vectors and the Gram-Schmidt coefficient matrix R, there are corresponding algorithms for the reduction of Gram matrices (cf., e. g., [Wet98]). The same holds for the reduction of singular Gram matrices which corresponds to the LLL reduction of arbitrary finite generating systems of a lattice.

It is trivial to modify $\mathsf{GenSample}$ such that the algorithm does not return the vector $\mathbf{v} \in V_{\mathbf{c},\mathsf{B}}$ anymore but rather $\mathbf{x} \in \mathbb{Z}^n$ such that $\mathsf{B}\mathbf{x} = \mathbf{v} \in V_{\mathbf{c},\mathsf{B}}$. Then $\mathsf{GenSample}$ operates – besides the search space parameter $\mathbf{c}$ and the index x – only on the Gram-Schmidt coefficient matrix R, i. e., data that the LLL reduction of a Gram matrix computes as well. The squared length of a sampled vector is given by $\|\mathbf{v}\|^2 = \mathbf{x}^t\mathsf{G}\mathbf{x}$ whence we can implement the sampling loop for Gram matrices.

After the sampling loop we need to construct the new (singular) Gram matrix G' corresponding to the hypothetical generating system $[\mathbf{v}, \mathsf{B}]$. Using $\mathbf{v} = \sum_{j=1}^n x_j\mathbf{b}_j$, we find

$$\mathsf{G}' = \begin{pmatrix} \|\mathbf{v}\|^2 & \mathbf{x}^t\mathsf{G} \\ \mathsf{G}\mathbf{x} & \mathsf{G} \end{pmatrix}$$

where $\mathsf{G} \in \mathbb{Z}^{n\times n}$ is the input Gram matrix.

This shows that the Sampling Reduction algorithms we described in this chapter can be easily transferred to the reduction of Gram matrices, even if we do not explicitly state the modified algorithms.

Chapter 4.

Quantum Lattice Reduction

Algorithms for the quantum computing model attracted a lot of interest in recent years, even though there are no implementations yet that can solve problems of practical significance. (See, e. g., [Gle05] for an impressive list of currently funded quantum computing projects.) Quantum computers can solve problems that are believed to be hard in the classical computing model. In particular, Shor [Sho97] proposed a remarkable quantum algorithm that solves the RSA problem in polynomial time. Hence, the eventual advent of quantum computers beyond the proof-of-concept stage will make it necessary to turn away from the worldwide most deployed cryptosystem. Recently, several research projects were launched that explore alternatives to the established cryptosystems that withstand attacks with quantum computers (see, e. g., [BCD+04]).

The literature has only very recent results on the hardness of lattice problems in the quantum computing model, most notably by Regev [Reg05, AR04, AR03, Reg02]. Those results that improve on classical results apply to special classes of lattices only [Reg03]. We describe in this chapter a variant of the Simple Sample Reduction algorithm – first presented in [Lud03] – that exploits the capabilities of quantum computers to boost the search of the sample space. It is to our knowledge the first result that indicates that quantum algorithms for general lattice problems asymptotically outperform classical algorithms. Our algorithm does *not* run in polynomial time, though.

4.1. Grover's Quantum Search

This section provides the background on Grover's quantum search required for an understanding of the Quantum Search Reduction algorithm proposed in the following section. We specify the pre- and postconditions of the quantum search algorithm and of the quantum counting algorithm which estimates the number of solutions to a search problem. We refer the reader to textbooks on the subject, e. g. [NC00], for more details.

The quantum computing model is based on the following four postulates of quantum mechanics:

State: Every *isolated* quantum system is associated with some Hilbert space V. The state of the quantum system is completely described by a unit length vector,

commonly written as so called *ket* vector $|\psi\rangle \in V$. Therefore, the state space of a quantum system is the unit sphere in its corresponding Hilbert space. In the context of quantum computing the associated Hilbert space is always $V = \mathbb{C}^N$ for some 2-power $N = 2^n$. In particular, the most simple quantum systems – the so called *qubits* – are associated with $V_q = \mathbb{C}^2$.

Composite Systems: Let two quantum systems be associated with the Hilbert spaces V_1 and V_2, respectively. Then the composite system is associated with the tensor space $V_1 \otimes V_2$. A quantum system built from k qubits is therefore associated with $V = \bigotimes_{j=1}^{k} V_q = V_q^{\otimes k} = \mathbb{C}^{2^k}$.

Evolution: Let $|\psi(t)\rangle \in V$ describe the state of a closed quantum system at time t, i. e., the state of a system that does not interact with its environment at all. For any $t_1, t_2 \in \mathbb{R}$, there is a unitary operator U on the associated Hilbert space V such that $|\psi(t_2)\rangle = U|\psi(t_2)\rangle$. Conversely, if $t_1 \neq t_2 \in \mathbb{R}$, $V = V_q^{\otimes n}$, and $\mathsf{U} \in \mathrm{End}(V)$ is unitary, then there is a closed quantum system associated with V such that $|\psi(t_2)\rangle = U|\psi(t_2)\rangle$.

Measurements: If the state of a quantum system is measured, then there is for any potential outcome $e \in E$ an operator $\mathsf{M}_e \in \mathrm{End}(V)$ such that $\sum_{e \in E} \mathsf{M}_e^\dagger \mathsf{M}_e = \mathsf{I}$. ($\mathsf{M}_e^\dagger$ is the Hermite conjugate or adjoint of M_e. I is the identity operator.) If the system is in state $|\psi\rangle$ before the measurement, then the probability of the outcome e is $\Pr[\, e \mid |\psi\rangle] = \langle\psi|\mathsf{M}_e^\dagger \mathsf{M}_e|\psi\rangle$ where the *bra* $\langle\psi| \in V^*$ is the dual of $|\psi\rangle \in V$. After a measurement with outcome e, the system is in the state $|\psi'\rangle = \Pr[\, e \mid |\psi\rangle]^{-1/2} \mathsf{M}_e|\psi\rangle$.

For our purposes, it suffices to consider measurements with respect to the canonical base vectors of $V = \mathbb{C}^{2^n}$, i. e., $E = \{\, 0, \ldots, 2^n - 1 \,\}$ and $M_e = |e\rangle\langle e|$ where $\{\, |0\rangle, \ldots, |2^n - 1\rangle \,\}$ is the (orthonormal) canonical basis of $\mathbb{C}^N$.

A quantum algorithm consists essentially of three steps: (a) We prepare an n qubit quantum register in a defined base state $|e\rangle$. (b) We apply some unitary operations on the register. (Note that afterwards the system is no longer necessarily in a base state but in some superposition $|\psi\rangle = \sum_{j=0}^{2^n-1} \psi_j |j\rangle$ subject to $\sum_{j=0}^{2^n-1} \overline{\psi_j}\psi_j = 1$.) (c) We measure the system and find it with probability $\overline{\psi_e}\psi_e$ and up to some unobservable phase factor $e^{\imath\varphi}$ in the base state $|e\rangle$, $e \in \{\, 0, \ldots, 2^n - 1 \,\}$. We can implement more complex algorithms if we measure subregisters only and base the choice of the subsequent unitary operations upon the outcomes.

As long as we do not measure any registers, the control flow in a quantum algorithm (i. e., the sequence of quantum operations applied to the register) can depend on the register size only, not on its current state. That is because the algorithm may be applied to a superposition of all possible (classical) inputs whence the quantum operations need to control virtually all possible threads of execution in parallel. That makes the computing model of *uniform circuit families* the most adequate to describe quantum algorithms.

Let $S_k = \{0,1\}^k$, $S_l = \{0,1\}^l$, and $f : S_k \to S_l$ a total function. Then

$$F : S_k \times S_l \to S_k \times S_l : (x,y) \mapsto (x, f(x) \oplus y)$$

is a bijection and $F^{-1} = F$. Hence, there is a unitary operator U_F on $k+l$ qubit registers such that $U_F(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |f(x) \oplus y\rangle$. In particular, measurement of $U_F(|x\rangle \otimes |0\rangle) = |x\rangle \otimes |f(x)\rangle$ yields with certainty the result $(x, f(x))$ whence U_F is said to compute f.

If f can be computed by a classical algorithm in worst case time $O(t)$ and on worst case space $O(m)$, then we can construct a sequence of $O(t)$ elementary unitary operations $\tilde{\mathsf{U}}_j$ (so called *gates*) operating on at most $O(m)$ additionally auxiliary qubits such that $\tilde{\mathsf{U}}_1 \circ \tilde{\mathsf{U}}_2 \ldots$ restricted to the first $k+l$ qubits is U_F. Thus, for any classical algorithm with worst case complexity in $O(f(n))$ we have an equivalent uniform quantum circuit family with circuit depth in $O(f(n))$ as well.

Theorem 25 (Grover; Boyer, Brassard, Hoyer, Tapp). *Let $S = \{0, \ldots, N-1\}$, $N = 2^n$. Let* Pred *be a classical algorithm that computes the predicate $\mathcal{P} : S \to \{0,1\}$.*

There is a quantum algorithm Grover *that, on input $M = |\mathcal{P}^{-1}(\{1\})| > 0$ and an n qubit quantum circuit $\mathsf{O}_\mathcal{P}$ such that $\mathsf{O}_\mathcal{P}|e\rangle = (-1)^{\mathcal{P}(e)}|e\rangle$ for any canonical base vector $|e\rangle$, returns some $s \in \mathcal{P}^{-1}(\{1\})$.* Grover *makes expected $\Theta((N/M)^{1/2})$ queries of $\mathsf{O}_\mathcal{P}$ and performs expected $\Theta((N/M)^{1/2})$ additional elementary quantum operations.*

This result is remarkable because in the classical computing model the search of an unordered set takes expected linear time in N/M whereas Grover is sublinear. Furthermore, this algorithm is known to be optimal; there cannot be a quantum algorithm for searching unordered sets that requires asymptotically less quantum gates. The black box oracle $\mathsf{O}_\mathcal{P}$ is essentially a quantum circuit that computes $\mathcal{P}$ and can therefore be algorithmically derived from the classical algorithm Pred.

Inspection of the Grover algorithm shows that M does not need to be exact, a good approximation is sufficient. The better the approximation, the less iterations of the probabilistic algorithm will be required on average. There is no impact on the asymptotic runtime as long as the error is less than, say, $M^{1/2}$.

The full proof of Theorem 25 can be found in [Gro96, BBHT96]. We briefly sketch the algorithm Grover for the case $0 < M \leq N/2$ that exhibits the most important parts. Set

$$|\alpha\rangle = \frac{1}{\sqrt{N-M}} \sum_{\substack{j=1 \\ \mathcal{P}(j)=0}}^{N} |j\rangle\,, \qquad |\beta\rangle = \frac{1}{\sqrt{M}} \sum_{\substack{j=1 \\ \mathcal{P}(j)=1}}^{N} |j\rangle\,,$$

and

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{j=1}^{N} |j\rangle\,.$$

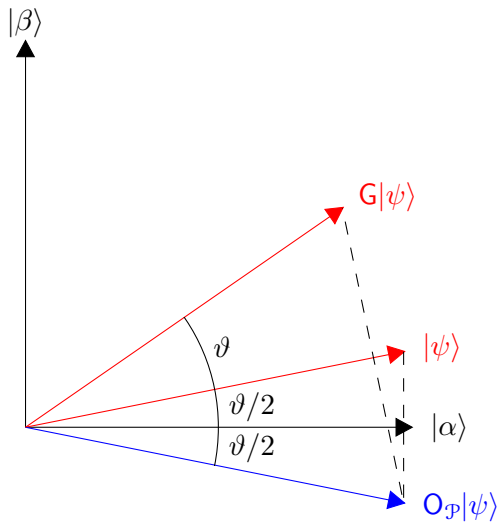


Figure 4.1.: The Grover rotation G (from [NC00])

$|\psi\rangle$ is the superposition of all search space elements, $|\alpha\rangle$ is the superposition of all non-solutions, and $|\beta\rangle$ is the superposition of all solutions. Observe that $|\psi\rangle = (M/N)^{1/2}|\beta\rangle + ((N-M)/N)^{1/2}|\alpha\rangle$ belongs to the subspace V' spanned by $|\alpha\rangle$ and $|\beta\rangle$.

The single qubit Hadamard gate H is defined by

$$\mathsf{H} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

and rotates the base vector $|0\rangle$ into the superposition $2^{-1/2}(|0\rangle + |1\rangle)$. The n qubit Hadamard operator is the n-fold tensor product $\mathsf{H}^{\otimes n} = \mathsf{H} \otimes \ldots \otimes \mathsf{H}$ of H with itself. $\mathsf{H}^{\otimes n}$ maps $|0\rangle$ to $|\psi\rangle$. Define the Grover operator

$$\begin{aligned} \mathsf{G} &= \mathsf{H}^{\otimes n}(2|0\rangle\langle 0| - \mathsf{I})\mathsf{H}^{\otimes n}\mathsf{O}_{\mathcal{P}} \\ &= (2|\psi\rangle\langle\psi| - \mathsf{I})\mathsf{O}_{\mathcal{P}}\,. \end{aligned}$$

We note (a) G can be realized without knowledge of $|\beta\rangle$ or M, (b) $O_{\mathcal{P}}$ restricted to V' is a reflection about $|\alpha\rangle$, and (c) $2|\psi\rangle\langle\psi| - \mathsf{I}$ restricted to V' is a reflection about $|\psi\rangle$. G has to be a rotation in V' since it is the composition of two reflections . Fig. 4.1 illustrates that the rotation is from $|\psi\rangle$ towards $|\beta\rangle$ by $\vartheta = 2\arcsin((M/N)^{1/2}) \in [0, \pi/2]$.

Thus, the Grover algorithm is as follows: We prepare first the register in state $|0\rangle$ and apply $\mathsf{H}^{\otimes n}$, so the register is in state $|\psi\rangle$. Then we apply k times the Grover operator G where $k = \lfloor(\pi - \vartheta)/(2\vartheta)\rceil \leq \lceil(\pi/4)(N/M)^{1/2}\rceil$, thereby transforming the register in a state near $|\beta\rangle$. Therefore, a final measurement of the register will find

it, with high probability, in a base state $|s\rangle$ that contributed to β, i. e., in a base state corresponding to a solution s. If the classical verification with Pred shows the unlikely event $\mathcal{P}(s) = 1$, then we simply restart Grover.

For the Grover algorithm to be useful we still need a way to estimate M or – equivalently – ϑ in advance. This is achieved by the quantum counting algorithm QC.

Theorem 26 (Boyer, Brassard, Hoyer, Tapp). *Let S, $\mathcal{P} : S \to \{0, 1\}$, $\mathsf{O}_{\mathcal{P}}$, N, and M as in Theorem 25. Let $\varepsilon > 0$ and $m \in \mathbb{N}_+$.*

There is a quantum algorithm QC that, on input the black box oracle $\mathsf{O}_{\mathcal{P}}$, returns, with probability at least $1 - \varepsilon$, the Grover rotation angle ϑ up to m binary digits precision. QC queries $\mathsf{O}_{\mathcal{P}}^{2^j}$ for all $j \in \{0, \ldots, t-1\}$ and requires $O(t^2)$ additional elementary quantum operations where $t = m + \lceil \log_2(2 + 1/(2\varepsilon)) \rceil$.

Since $\sin^2(\vartheta/2) = M/N$, we can derive an estimate $\tilde{M}$ for the number of solutions in S from the value $\tilde{\vartheta}$ returned by QC. For example, if $t = \lceil n/2 \rceil + 3$ than we have $|M - \tilde{M}| \in O(M^{1/2})$.

ϑ is the angular velocity of the map $t \mapsto \mathsf{G}^t|\psi\rangle$ whence it is no surprise that ϑ can be recovered from some samples $\mathsf{G}|\psi\rangle, \mathsf{G}^2|\psi\rangle, \ldots$ by means of the Fourier transform. The details of Shor's renowned quantum Fourier transform [Sho94] are beyond the scope of this outline of quantum searching. Suffice it to say that the quantum Fourier transform in the shape of the so called phase estimation algorithm computes ϑ up to t binary digits from $\mathsf{G}^{2^0}|\psi\rangle, \ldots, \mathsf{G}^{2^{t-1}}|\psi\rangle$. The QC algorithm was first described in [BBHT96] and later refined in [BHT98].

4.2. The Quantum Search Reduction Algorithm

We describe in this section a quantum algorithm for the reduction of lattice bases. This algorithm is a variation of SSR using Grover's search algorithm and the quantum counting algorithm.

SSR has two parts that involve a search problem in the set $S_{u,\mathsf{B}}$ or – using the generalizations from Sect. 3.7 – in the set $V_{\mathbf{c},\mathsf{B}}$: CSSS returns true if and only if the implementation deems it likely that there is at least one solution $\mathbf{v} \in V_{\mathbf{c},\mathsf{B}}$ subject to $\|\mathbf{v}\|^2 \leq \gamma \|\mathbf{b}_1\|^2$. The sampling loop in SSR systematically searches $V_{\mathbf{c},\mathsf{B}}$ for such a solution. The *Quantum Search Reduction* QSR on the following page replaces CSSS and the sampling loop by QC and Grover, respectively.

There is the minor technical inconvenience that both quantum algorithms expect a quantum circuit $\mathsf{O}_{\mathcal{P}}$ as black box oracle and the predicate evaluated by this oracle changes every outer SSR loop iteration. The workaround is to store the parameters the predicate depends on in some extra quantum registers that remain in some fixed base state throughout each QSR iteration. In particular, the extra registers won't be entangled with the working registers of QC and Grover and will therefore not interfere with these algorithms. More descriptive, the black box oracle uses the extra registers like global read-only variables in a classical algorithm.

Algorithm 10 QSR

Input:
- basis $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{d \times n}$ of lattice L
- reduction factor $\gamma \in (0, 1)$,
- $\mathbf{c} \in \mathbb{N}^n$ as in Def. 16,
- LLL parameter $\delta \in (1/4, 1)$

Output: LLL reduced basis B' of L such that $\gamma \|\mathbf{b}'_1\|^2 < \min\{ \|\mathbf{v}\|^2 \mid \mathbf{v} \in V_{\mathbf{c},\mathsf{B}'} \}$.

/* m: floating point size, m': small constant in $\mathbb{N}_+$ */

prepare $|r_{\mathfrak{b}}\rangle|r_{\mathsf{R}}\rangle|r_\gamma\rangle|r_{\mathbf{c}}\rangle|r_x\rangle \in V_q^{\otimes mn} \otimes V_q^{\otimes mn(n-1)/2} \otimes V_q^{\otimes m} \otimes V_q^{\otimes m'n} \otimes V_q^{\otimes \log_2 |V_{\mathbf{c},\mathsf{B}}|}$
$|r_\gamma\rangle|r_{\mathbf{c}}\rangle \leftarrow |\gamma\rangle|\mathbf{c}\rangle$
$\mathsf{O}_{\mathcal{P}} \leftarrow \mathsf{QuantumCircuit}(x \mapsto \mathsf{SamplePredicate}(\mathfrak{b}, \mathsf{R}, \gamma, \mathbf{c}, x))$
$(\mathsf{B}, \mathfrak{b}, \mathsf{R}) \leftarrow \mathsf{LLL}(\mathsf{B}, \delta)$ /* $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$, $\mathfrak{b} = (\|\hat{\mathbf{b}}_1\|^2, \ldots, \|\hat{\mathbf{b}}_n\|^2)$ */
$|r_{\mathfrak{b}}\rangle|r_{\mathsf{R}}\rangle \leftarrow |\mathfrak{b}\rangle|\mathsf{R}\rangle$
$M \leftarrow \mathsf{QC}(\mathsf{O}_{\mathcal{P}})$
while $M > 0$ **do**
  $x \leftarrow \mathsf{Grover}(\mathsf{O}_{\mathcal{P}})$
  $\mathbf{v} \leftarrow \mathsf{GenSample}(\mathsf{B}, \mathsf{R}, \mathbf{c}, x)$
  $(\mathsf{B}, \mathfrak{b}, \mathsf{R}) \leftarrow \mathsf{LLL}([\mathbf{v}, \mathbf{b}_1, \ldots, \mathbf{b}_n], \delta)$
  $|r_{\mathfrak{b}}\rangle|r_{\mathsf{R}}\rangle \leftarrow |\mathfrak{b}\rangle|\mathsf{R}\rangle$
  $M \leftarrow \mathsf{QC}(\mathsf{O}_{\mathcal{P}})$
end while

However, this implies that we know the space occupied by the global black box parameters in advance, for all QSR iterations. The algorithm Sample used in the search condition above refers to the lattice basis B and computes a lattice vector $\mathbf{v}$. While it is possible to come up with space bounds for all B and $\mathbf{v}$ that may be computed in the course of QSR, these are in fact worst case bounds. Since quantum space is a scarce resource, we'd like to avoid any waste of qubits. On inspection of the criterion $\|\mathbf{v}\|^2 = \|\mathsf{GenSample}(\mathsf{B}, \mathsf{R}, \mathbf{c}, x)\|^2 \leq \gamma \|\mathbf{b}_1\|^2$, we realize that we do not need to make $\mathbf{v}$ explicit as long as we can compute its norm. We can easily reconstruct the short vector $\mathbf{v}$ we want to prepend to the basis from its index in $V_{\mathbf{c},\mathsf{B}}$ returned by Grover. This idea is implemented by the algorithm SamplePredicate on page 66.

Therefore, QSR can be outlined as follows: The algorithm first prepares all quantum registers required by the black box and initializes the registers that will not change. Then it determines the quantum circuit $\mathsf{O}_{\mathcal{P}}$ such that it fits the search space size $|V_{\mathbf{c},\mathsf{B}}|$. (As mentioned in Sec. 4.1, there is a uniform quantum circuit family of suited black box oracles with circuit depth in $O(n^2)$ because SamplePredicate is classically computable in $O(n^2)$ time. Furthermore, uniform circuit families are inherently constructive, so QSR can actually fix the required circuit.) Then, as long as QC asserts there is indeed a sufficiently short vector in $V_{\mathbf{c},\mathsf{B}}$, QSR employs Grover to find the index of such a vector $\mathbf{v}$, reconstructs $\mathbf{v}$ by means of GenSample, and finally

LLL reduces the generating system formed by prepending $\mathbf{v}$ to B.

How many qubits need to be reserved for the extra registers $\mathsf{O}_{\mathcal{P}}$ operates on? SamplePredicate expects the squared norms $\mathfrak{b}$ of the Gram-Schmidt vectors of B, the Gram-Schmidt coefficient matrix R, the reduction factor γ, the search space descriptor $\mathbf{c}$, and the search space element number x. All but the last parameter need to be stored in extra registers prior to the calls of QC and Grover. The Schnorr-Euchner variant of LLL, which is mostly used in practice, as well as BKZ store $\mathfrak{b}$ and R in floating point format, so we can assume a floating point format here as well. Let m be the number of Bits occupied by each such floating point number. Then the extra register that holds $\mathfrak{b}$ has to be mn qubits wide. The upper triangular matrix R requires a $mn(n-1)/2$ qubit register. The product of all nonzero entries in $\mathbf{c}$ is the search space size $|V_{\mathbf{c},\mathsf{B}}|$. Since the search in a set with more than 2^{160} elements seems infeasible, even with Grover, $m'n$ qubits (with some small constant $m' = 4$, say) are certainly enough to store $\mathbf{c}$.

The algorithm SamplePredicate follows the path set forth by GenSample. It exploits that GenSample never references the value of $\mathbf{v}$. So the algorithm still computes $\boldsymbol{\nu}$ such that $\mathsf{Sample}(\mathsf{B}, \mathsf{R}, x) = \hat{\mathsf{B}}\boldsymbol{\nu}$ if we remove all occurrences of $\mathbf{v}$. Then the input parameter B is not used anymore whence we can remove it as well.

Recall that $\mathbf{c}$ contains trailing zeros only. SamplePredicate thus executes the first branch of the innermost **if** block exactly once. All following iterations execute the **else** branch and modify only coefficients ν_i of $\boldsymbol{\nu}$ with $i \leq j$ because R is unit upper triangular. This implies that the update after the innermost **if** block suffices to restitute the loop invariant $\ell = \|\sum_{i=j+1}^{n} \nu_i \hat{\mathbf{b}}_i\|^2$. Hence, $\ell = \|\mathsf{Sample}(\mathsf{B}, \mathsf{R}, x)\|^2$ at the end of the algorithm. (Note that the right hand side is invariant if we replace B by any real matrix B' with Gram-Schmidt decomposition $\mathsf{B}' = [\hat{\mathbf{b}}'_1, \ldots, \hat{\mathbf{b}}'_n]\mathsf{R}$ such that $\|\hat{\mathbf{b}}'_i\|^2 = \|\hat{\mathbf{b}}_j\|^2$ for all $j \in \{1, \ldots, n\}$.) SamplePredicate can therefore test the inequation $\|\mathsf{Sample}(\mathsf{B}, \mathsf{R}, x)\|^2 \leq \gamma\|\mathbf{b}_1\|^2$ by comparing ℓ with the scaled value of $\|\hat{\mathbf{b}}_1\|^2 = \|\mathbf{b}_1\|^2$.

4.3. QSR vs. Classical Algorithms

We cannot construct a quantum variant of PoolSR because Grover returns exactly one index into the search space and the predicate needs to be deterministic. We cannot express "almost short enough" within the Grover framework. We can formulate such a predicate for the vectors sought in ShortProjectionSR, though, whence it is possible to construct a quantum variant of ShortProjectionSR in analogy to QSR.

Due to the nature of quantum measurements, Grover returns a random solution to the search problem specified by $\mathsf{O}_{\mathcal{P}}$. QSR is thus a randomized algorithm, different from the deterministic SSR.

Let M be the number of sufficiently short vectors in $V_{\mathbf{c},\mathsf{B}}$ and $|V_{\mathbf{c},\mathsf{B}}| = 2^u$. SSR needs expected $O(n^2 2^u/M)$ classical operations to find a vector it can insert in front of $\mathbf{b}_1$. Grover, on the other hand, comes up with such a vector after only $O(n^2(2^u/M)^{1/2})$ quantum operations.

Algorithm 11 SamplePredicate

Input:
- $\mathfrak{b} = (\|\hat{\mathbf{b}}_1\|^2, \ldots, \|\hat{\mathbf{b}}_n\|^2) \in \mathbb{Q}^n$
- unit upper triangular matrix $\mathsf{R} = [\mathbf{r}_1, \ldots, \mathbf{r}_n] \in \mathbb{Q}^{n \times n}$ such that there is $\mathsf{B} \in \mathbb{Z}^{d \times n}$ with Gram-Schmidt decomposition $B = [\hat{\mathbf{b}}_1, \ldots, \hat{\mathbf{b}}_n]R$,
- reduction factor $\gamma \in (0, 1)$,
- $\mathbf{c} \in \mathbb{N}^n$ as in Def. 16,
- $x \in \{0, \ldots, |V_{\mathbf{c},\mathsf{B}}| - 1\}$

Output: 1 if $\|\mathsf{GenSample}(\mathsf{B}, \mathsf{R}, \mathbf{c}, x)\|^2 \leq \gamma \|\mathbf{b}_1\|^2$ and 0 else.

```
procedure SamplePredicate (𝔟, R, x)
    (ℓ, ν) ← (0, 0)
    for j from n downto 1 do
        if c_j > 0 then
            (x, z) ← (⌊x/c_j⌋, x mod c_j)
            if j = n or c_{j+1} = 0 then
                ν ← (x + 1)r_j          /* ν = (ν_1, ..., ν_n)^t,  ν_{j+1} = ··· = ν_n = 0 */
            else
                z ← (−1)^z ⌊z/2⌋                          /* z ∈ ℤ ∩ (−c_j/2, c_j/2] */
                y ← ⌈ν_j − 1/2⌉                           /* ν_j − y ∈ (−1/2, 1/2] */
                if ν_j − y ≤ 0 then
                    y ← y − z                             /* ν_j − y ∈ (−c_j/2, c_j/2] */
                else
                    y ← y + z                             /* ν_j − y ∈ (−c_j/2, c_j/2] */
                end if
                ν ← ν − y r_j                             /* ν_j ← ν_j − y */
            end if
            ℓ ← ℓ + ν_j^2 ‖b̂_j‖^2
        end if
    end for
    if ℓ ≤ γ‖b̂_1‖^2 then
        return 1
    else
        return 0
    end if
end procedure
```

Furthermore, the output precision of QC needs to be sufficiently high for us to decide $M > 0$ or not. A more detailed analysis shows that QC answers this question if ϑ is determined up to $m = \lceil u/2 \rceil + 1$ bit precision [NC00]. In that case, QC performs $\Theta(2^{u/2})$ Grover iterations. The runtime of QSR is hence dominated by QC and the total cost of a single iteration is $O(n^2 2^{u/2} + n^3)$ operations. In contrast, a single SSR iteration requires $O(n^2 2^u / M + n^3)$ operations. QSR is thus asymptotically

faster than SSR provided $M \in o(2^{u/2})$ which, empirically, we can assume after a few iterations, at the latest. The number of samples we have to compute before a short vector is found usually grows rapidly whence QSR offers an almost quadratic speedup – asymptotically, that is.

It is not possible yet to predict the actual runtime that will be attached to a single operation on a quantum computer. It is not even clear yet which technology will be used to realize "large" quantum computers. It is therefore pointless to try to compare the O-constants in the runtimes of QSR and SSR. If, hypothetically, Grover will in fact realize a quadratic speedup with respect to the wall clock time required, then QSR will enable us to search a candidate set in about two weeks that is infeasible to search with today's classical hardware: Our primary test hardware enumerates approximately 2^{33} sample vectors in a $n = 180$ dimensional lattice per fortnight. Under our hypothesis, a quantum computer will therefore search up to 2^{66} sample vectors within the same time span. Adding n^2 operations per sample for the evaluation of SamplePredicate, the classical search of that many vectors would cost more than 2^{80} operations.

Another play of thought considers the SVP approximation factor achieved by QSR if we return to (GSA). Then Schnorr's analysis of RSR applies and RSR computes in $O(n^3 e^{(k/4)\ln(k/6)} + n^4)$ time a vector at most $e^{(n/2k)\ln(k/6)}$ times as long as a shortest lattice vector. If we replace the random sampling in RSR by Grover in analogy to QSR then the runtime drops to $O(n^3 e^{(k/8)\ln(k/6)} + n^4)$. Or, equivalently, a quantum variant of RSR achieves in $O(n^3 e^{(k/4)\ln(k/3)} + n^4)$ time an SVP approximation factor as small as $e^{(n/4k)\ln(k/3)}$; i. e., the approximation factor is improved almost quadratically.

Chapter 5.

The LaRed Framework

We describe LaRed, a C++ library and framework for lattice basis reduction by sampling. The design of LaRed eases the implementation of different sampling reduction strategies and allows for the use of almost arbitrary long integer and floating point arithmetic packages. We also describe the Python extension module laredpy that supports the interactive use of LaRed and provides tools for the evaluation of algorithms implemented with LaRed.

5.1. LaRed

In Chapter 3, we reported on the empirical behaviour of the proposed algorithms. For this we needed an implementation of various sampling reduction algorithms until the algorithms reached the form presented in this thesis. It soon became clear that ad-hoc prototypes did not offer the flexibilty we needed. We therefore developed a framework and accompanying library named LaRed for lattice basis reduction by sampling.

We motivate the requirements that governed the development of LaRed in the following Sect. 5.1.1. Then Sect. 5.1.2 explains our design choices for LaRed in order to meet these requirements, before Sect. 5.1.3 provides the information about LaRed's implementation that is needed for its effective use.

5.1.1. Design Goals and Requirements

If LaRed is to be used beyond this thesis it has to offer an intuitive interface and has to make efficient use of the available computing resources. It has to be portable so it can be built and behaves correctly not only today on different platforms but on future platforms as well. Given the rapid change of affordable computer hardware this portability serves also LaRed's maintainability.

We have seen that Sampling Reduction is in fact a family of algorithms that follows the pattern described in Alg. 12. There are many reasonable ways to implememt this pattern: The LLL-type reduction can be, e. g., the Schnorr-Euchner variant of LLL, BKZ, or Koy's primal-dual reduction. We discussed several possible implementations of the loop condition in Sect. 3.2 through 3.4. We can sample vectors from the search space at random or by enumeration. The selection of sample vectors used to form the new generating system may depend on $\mathbf{b}_1$ only or on the Gram-Schmidt vectors

Algorithm 12 Sampling Reduction Pattern

Input: generating system G of a some latticce L
Output: reduced basis B of L

```
B ← LLL-type reduction of B
while chance of progress considered sufficient do
    Sample vectors from some search space and recombine some of them with B
        into a new generating system G of L = L(B).
    B ← LLL-type reduction of G
end while
return B
```

$\hat{\mathbf{b}}_j$ as well, and so on. All these variations have in common that they do not interfere with each other; e. g., we can choose the method to sample vectors regardless of the LLL-type reduction. In other words, Sampling Reduction is the composition of several orthogonal policies.

From the outset, our goal was that LaRed is easily extendible due to a thorough separation of concerns [HL95]. LaRed should impose a minimal interface only on the policies, so a user can easily define her own policy implementations which substitute the policies LaRed ships with. Ideally, policy modifications should still be possible at runtime so as not to discourage experimentation. This implies that code changes due to a modified policy should be as locally confined as possible. It should not be necessary to alter any code that is not part of the respective policy's implementation.

We therefore concretize the term *policy* as a piece of code that, as Alexandrescu [Ale01] puts it, "takes care of only one behavioral or structural aspect. As the name suggests, a policy establishes an interface pertaining to a specific issue. You can implement policies in various ways as long as you respect the policy interface." In contrast to Alexandrescu, though, we do not require a policy to be applied at compile time.

As mentioned in Sect. 3.7.2, the search of $V_{\mathbf{c},\mathsf{B}}$ for a short vector is an application where SIMD-parallelization (single instruction, multiple data) excels. The use of software packages like DOTS [BKLW99] can be encapsulated into policies. However, in the presence of concurrency, any non-const global or static data poses a risk of race conditions, though. Since we don't want to preclude the use of such a distributed search policy, LaRed must not use non-const global or static data.

Policies typically modify the behaviour of an algorithm. There is another aspect of LaRed that has to be configurable but that ideally does not change the behaviour of the implemented algorithms: The long integer and floating point arithmetic used. In particular, the inherent limitations of floating point arithmetic (FPA) imply that every choice of FPA is a compromise between the lattice bases an algorithm can actually reduce and its efficiency. For instance, the Gram-Schmidt decomposition of an input basis can fail if the ratio between the maximum and minimum length of the Gram-Schmidt vectors becomes too large and in consequence there are too few precise

digits left in the Gram-Schmidt coefficients. In such a case, a more precise (software) FPA type is required; on the other hand, hardware FPA – e. g., the elementary C++ type `double` – is typically more efficient by at least an order of magnitude. LaRed therefore has to support various arithmetic types.

However, code duplication hurts maintainability and should be avoided where possible. It follows that LaRed's arithmetic should be configurable by means of type traits. By *type trait* we refer to a mechanism that "establishes a uniform symbolic interface over a coherent set of design choices that vary from one type to another" [Ale00]. As an additional benefit, the freedom in the choice of arithmetic enables us to interface LaRed to existing computational algebra libraries like LiDIA [LG04] and NTL [Sho05].

The boundaries between policies and type traits are often somewhat fuzzy; as a rule of thumb, policies let the user customize how some objective is achieved – they fill in the respective algorithms. Type traits, in contrast, provide the information required to instantiate a generic implementation of some algorithm for different data types that model a common concept.

Early on we encountered a further requirement: Experiments with high dimensional lattice bases often ran several days whence they were impractical to perform on a notebook that gets turned off once or twice a day. Even on a stationary PC, we had to abort long running experiments due to security critical operating system updates, power outages and so on. The system thus had to support dumping the algorithm's current state in regular intervals to disk and resuming the reduction later on starting in this state.

Finally, some reductions are best run under interactive control – as seen, e. g., in Sect. 3.6.2. It was therefore necessary to develop a convenient interface that supports the available policies and does not make the definition of new interfaces prohibitively hard.

5.1.2. Design Choices

While testing several prototypes of LaRed, the above mentioned requirements became apparent. We discuss in this section some of the corresponding design alternatives and describe the choices we finally settled on.

Policies

A traditional approach to place some decisions within an algorithm in the user's responsibility is through callback functions. For instance, the LLL implementations in NTL expect a function pointer `check` as parameter. If `check`= 0! then it refers to a function that is called in each iteration with the base vector last computed as argument. The LLL-reduction is terminated immediately if `*check` returns a nonzero value because, say, the vector is short enough for a given application.

Plain function pointers are rarely used in modern C++ libraries. The main reason is that functions are not stateful – non-const function scope static data aside which is

considered harmful in general because it is prone to break the system in the presence of concurrency. Following the lead of the C++ standard library, function pointers are often replaced by so called *functors*, i. e., objects of a class type for which `operator()` has been overloaded. Functors are called with the funtion call syntax we know from function pointers as well, but they can be parameterized when they are constructed and they can keep state between calls. Actually, there is nothing special about `operator()`; any member function can assume its role if there is a convention within the framework how this member function is named. The objects that replace plain function pointers must implement a given interface; the semantics they implement may vary within certain bounds. In other words, these objects must be substitutable and the operations they define have to be polymorphic as defined by Liskov [Lis88].

There are two realizations of polymorphism in C++, static and dynamic. The former is also known as compile time polymorphism because it is implemented by means of class and function templates. As long as they support the same syntax, objects of different types can be treated the same way; the compiler can figure out which functions need to be actually called. The advantage of static polymorphism is that, at this stage, we still can replace function signatures and the types the algorithms operate on. These manipulations can be arbitrary complex; in fact, the C++ template mechanism offers us a Turing complete language [AG04] – the programs implemented in this language operate on types and integral numbers and their results, when executed by the compiler, are C++ code as well.

Static polymorphism is surprisingly powerful. But it has three disadvantages that go against the design goals mentioned in Sect. 5.1.1: Template metaprogramming is not a feature C++ was designed for; it was merely "discovered" [Unr02] and uses a rather awkward syntax. Therefore, programs using static polymorphism tend to be not easily accessible and, in consequence, hard to maintain. Abundant use of templates stresses the compiler, often to its limits, and even recent compilers are known to fail to implement templates conforming to the C++ standard. Porting code that relies on non-trivial static polymorphism often requires to work around those compiler deficiencies. Static polymorphism can thus hurt portability. These two drawbacks have to be weighed whenever static polymorphism is a reasonable design alternative; still, static polymorphism is often succesfully used. But our goal that policies can be easily replaced, ideally at runtime, tipped the scale since static polymorphism is inherently compile time bound.

So we are left with dynamic polymorphism, realized in C++ by means of virtual member functions. The design of policies based on runtime polymorphism is in fact an instance of the Strategy design pattern [GHJV95]. That is, for each policy we need a strategy class that defines the interface. Any member function of the strategy class that is intended to be customized by user defined policies is declared (pure) virtual. At runtime, we supply pointers or references to objects of derived types which implement the complete interface. The concrete policy types may be defined outside LaRed and the applied policy can be changed between calls into LaRed by simply exchanging the referenced policy objects. As we will see below, we can even

define new policy types at runtime through laredpy.

Dynamic polymorphism is not without drawbacks either: A virtual member function call is more time expensive and constitutes in most cases a barrier for any compiler optimization. This hurts most if the member function implementation is simple enough to be inlined if called statically. But we decided to accept a potential performance penalty – few percents in the worst case – in exchange for the flexibility we gain if we can easily switch policies.

Type Traits

We explained above that LaRed should support different arithmetic data types. The arithmetic lies at the core of LaRed's algorithms whence we could not as easily accept a performance hit encurred by dynamic polymorphism as with the policies above. On the other hand, it seemes likely that a user will use a small set of arithmetic types only. It is feasible and practicable to instantiate the algorithms and policies for all arithmetic types the user plans to use. Therefore, the arithmetic in LaRed is realized by means of static polymorphism.

More precisely, the main algorithm classes are parameterized on the long integer type used to represent the elements of the input basis and on the floating point type used for the Gram-Schmidt decomposition of the basis. These classes make both types accessible through public member typedefs `Integer` and `FP`, respectively. Then all policy classes are parameterized on the calling algorithm class. That way there is only one template parameter; the code does not become too complicated and unmaintainable due to the cumbersome template syntax.

Of course, we must actually operate on objects of the respective arithmetic types. We could reasonably expect that these overload the arithmetic operators. But we also need to compute logarithms, square roots, truncate floating point numbers and so on. There is no generally agreed upon arithmetic type concept that covers these function names. Furthermore, assume we required that, for every floating point object `x` of type `T`, the expression `sqrt(x)` returns the square root of `x` as another object of type `T`. Then all calls of `sqrt` from within LaRed had to be unqualified and we had to trust the compiler to find the correct function by argument dependent lookup (ADL, also known as Koenig lookup). That is because the functions for different arithmetic types are defined in different namespaces, and we have no other way to tell the compiler to look up `sqrt` in a namespace the name of which we do not know in advance and that does not enclose LaRed's namespace. Unfortunately, such use of ADL is rather fragile; for instance, the outcome depends on whether the compiler implements the two phase name lookup required by the C++ standard – which some compilers, that are the default compilers on many systems, still do not (GCC 3.3, MSVC++ 7.1, among others).

We overcome this by an indirection that the compiler can easily optimize away: We declare all required functions as function templates in a separate, nested namespace `LaRed::arithmetic`. These function templates simply forward the calls to type specific implementations. We provide inlined definitions of the primary templates

that rely on ADL as above. As long as the corresponding function for a particular arithmetic type is not named different from our default and as long as the compiler is able to resolve the function name by ADL, the primary templates are sufficient; the user does not need to write any support code. Otherwise, the user only hass to provide explicit specializations of the respective function templates that forward the calls through qualified function calls.

We also need some type traits that map from the arithmetic types to some constants. This is easily achieved by static member variables of class templates. For example, `LaRed::arithmetic::Zero<T>::value` is supposed to be a constant of type `T` with value 0. Again, the primary templates' definitions should fit all types that have conversion constructors accepting `int`s. If objects of this type cannot be initialized this way or the compiler used is not up to it, then an explicit specialization is an easy workaround.

State Serialization

We mentioned in Sect. 5.1.1 the need to store the current reduction's state in non-volatile memory combined with the ability to resume the reduction from the stored state later on. We first had to decide how to trigger a dump of the current state.

On Unix and similar systems it is common to make deamons reload their configuration or log statistics by sending some signal. We discarded the option to let users of LaRed request a state dump at more or less arbitrary points in the reduction by a similar mechanism. The handling of such asynchronous events would have involved low level system programming and thus hurt both portability and maintainability of LaRed for a doubtful gain.

We rather opted for a synchronous solution, i. e., the current state can be written on few well defined points only. For simplicity's sake, we decided that a state dump should be triggered whenever a user defined time interval has passed.

Note that the various reduction algorithms are not independent: For instance, BKZ calls LLL as a subalgorithm. In a design that relies on the language inherent stack mechanism to call subalgorithms, we still could serialize the state of all active algorithms – by always passing a pointer to the "parent" algorithm object, say, so we could walk up the stack and serialize all reduction objects. However, there is no portable way to restore an arbitrary deep process stack when a reduction dumped to disk is to be resumed.

We therefore had to include our own stack of reduction algorithms in LaRed. When resuming a reduction, this stack is reconstructed and the algorithm pushed last on the stack is executed. This also means that the algorithm implementations need to maintain some explicit information what part of the algorithm needs to be executed next so they can proceed when called after stack reconstruction. In other words, the algorithms have to be implemented as some kind of finite state machines.

In retrospect, the requirement to be able to dump and resume reductions had a major impact on LaRed's design – perhaps much more than one would be willing to grant, on the first glance, to a feature not directly related to LaRed's objective,

lattice basis reduction. However, it turned out that the algorithms' decomposition when implementing the state machines helped with identifying separating potential policies. Thus, the implementation became actually less intertwined and more clear.

Interface

It is easy to define free function templates that shield the user from the handling of LaRed's own stack and the setup of the algorithm classes, which are mostly implementation details. These function templates can provide default policies if the user chooses not to supply her own.

Since the abstract policy classes are parameterized on the algorithm class specializations, the algorithm classes necessarily leak into the user code. The impact on the maintainability of the user code can be minimized, though, by providing typedefs for the strategy class specializations. If only a fixed set of arithmetic types needs to be supported, then we can also provide an extra layer that hides the template specializations. For instance, LaRed ships with an NTL interface that supports all algorithms with the long integer type `NTL::ZZ` combined with the floating point types `double`, `NTL::quad_float`, `NTL::xdouble`, and `NTL::RR`, respectively. The user of this interface does not need to be aware that all policy classes are in fact template classes.

The user is free to change policies after each sampling reduction iteration because we chose not to implement the outer loop within LaRed. There is rather an algorithm class that runs CSSS and computes the new generating system G using the user defined policies. It is up to the user to reduce G again by an LLL-type reduction of her choice. So the user can apply in each iteration any defined policy.

Implementing an interactive user interface from scratch that puts the user in complete control of the reductions run is a major undertaking. We avoided that by defining a binding to the interpreted programming language Python. Within the Python interpreter, we can run the reductions with arbitrary complex policy combinations. Incidentally, the binding even supports that we derive Python classes from the C++ strategy classes, whence we can define new policies at runtime. The Python binding is described in more detail in Sect. 5.2.

5.1.3. Implementation

The previous section discussed LaRed on a more abstract level. Here we describe the actual implementation of LaRed. After some remarks about the code LaRed is based on, we will first see how `AlgorithmStack` controls the execution of the algorithms and how this interacts with the serialization and resumption of reductions. Then we give an overview over the sampling reduction policies already implemented in LaRed and how they interact.

The implementation of LLL and BKZ in LaRed evolved from code in NTL. NTL is distributed under the GNU General Public License (GPL). We could therefore adapt the code, but, in concequence, LaRed is covered by the GPL as well.

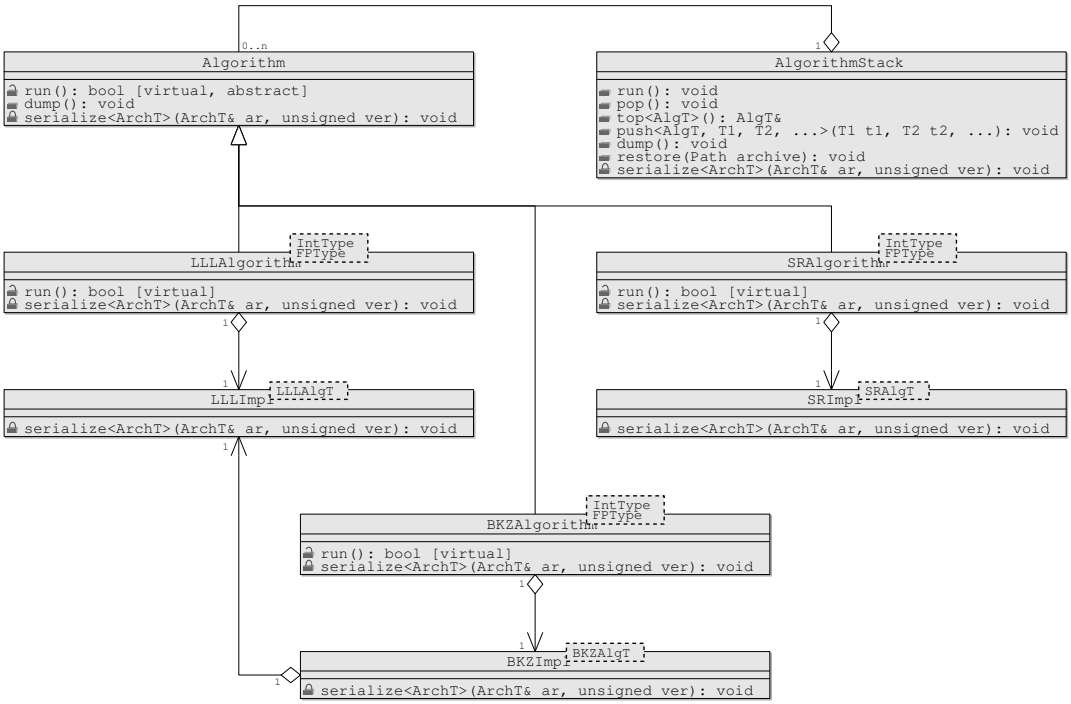


Figure 5.1.: Algorithm Classes (◊ aggregation, △ inheritance)

LaRed makes heavy usage of some Boost libraries [Boo04]. Boost is a collection of free peer reviewed C++ source libraries. It was started to provide reference implementations for potential additions to the C++ standard library and to establish existing practice prior to standardization. Boost therefore promotes quality programming practices and strives for maximum portability. The implementation of LaRed relies on six Boost libraries in particular: Boost.Smart Pointer automates memory management by means of reference counting. Boost.Serialization handles the writing and reading of complex data structures to and from disk. This involves some interaction with the filesystem; because of Boost.Filesystem we can avoid any operating system specific file system API. Boost.Preprocessor supplies the tools needed to implement variadic forwarding function templates. Boost.Random implements (non-cryptographic) pseudo random number generators of which we use the Mersenne Twister generator [MN98]. Boost.Test, finally, provides the framework for the unit tests that we implemented to avoid regressions.

Unless mentioned otherwise, all code that is part of LaRed's public interface is placed in the namespace `::LaRed`.

The relationship between the algorithm classes and the execution stack is shown in Fig. 5.1. The diagram uses an UML-like notation and shows an selection of the respective class members only.

An `AlgorithmStack` contains `Algorithm` objects. These objects can be accessed through a typical stack interface: `pop()` removes the topmost object from the stack and destroys it. Given a non-empty stack, a reference to the topmost element can be obtained by `top()`. If it is known that the top most object can be cast to the type `AlgT`, then it is possible to make `top()` return a reference to this type by use of the explicit template syntax `top<AlgT>()`. This function template enforces type safety; any illegal cast causes an exception to be thrown at runtime. The only way to push a new element onto the stack is through the member function template of the same name. Note, however, that it does not push an already existing object onto the stack; you rather have to provide the object's type `AlgT` and a parameter list. `push<AlgT>(...)` constructs a new `AlgT` object on the free store, forwarding the supplied parameter list to the appropriate constructor. Due to this restriction, an `AlgorithmStack` can enforce its ownership of all `Algorithm` objects it contains. This encapsulation of responsibilities avoids potential interactions with code that was not written for the LaRed framework and thus improves its maintainability. The constructor forwarding is implemented with the help of the Boost Preprocessor library by horizontal preprocessor iteration, a technique well known in the field of C++ generative programming, see, e. g., [AG04]. By default, LaRed supports up to twenty arguments which we suppose to serve almost any conceivable algorithm class. If not, then the limit can still be pushed further by a redefinition of the preprocessor constant `LARED_ALGORITHM_MAX_ARITY`.

A reduction's current state is defined by the state of all elements in the object graph that is accessible from the corresponding `AlgorithmStack`. When we write this state to disk we need to visit the objects in this graph, store their data members, and proceed with all objects they reference. We also need to recognize if we already stored an object before, so as not to enter an infinite recursion. If an object's data was already serialized, then we have to note where in the previous data stream the relevant information can be looked up. Conversely, when we read the serialized graph, we must ensure that no object is deserialized twice and that references to the same object are restored correctly – even if the object is referenced through pointers to base classes.

This is anything but a trivial task, but the complexity involved is encased in the Boost Serialization library. We only had to write some boilerplate code for any class that may be serialized. With this in place, we can dump the complete object graph reachable through the stack object `s` by an invocation of `s.dump()`. This member function makes sure that at least the time span `s.dump_interval` passed since the last dump. If so, then it creates a backup of the the previous archive file, opens a new archive, and writes `s` with a single line of code into the archive – together with all `Algorithm` objects `s` contains as well as all directly or indirectly referenced objects.

The reconstruction of a reduction is even simpler: Given the name of an archive file, `s.restore(name)` deserializes all objects stored in the archive and resets all references such that the new object graph is equivalent to the graph that was dumped into the file `name`.

```
void
AlgorithmStack::
run() {
  while(!this->empty()) {
    this->dump();
    if(this->top<Algorithm>().run()) {
      this->pop();
    }
  }
}
```

Listing 5.1: `AlgorithmStack::run()`

We indicated in Sect. 5.1.2 that the algorithms need to be implemented as state machines of some kind. Their input is the generating system, the Gram-Schmidt decomposition computed so far, etc. Let `a` be an `Algorithm` object. Each possible state of `a` corresponds to a particular section of the algorithm `a` implements. Then `a.run()` executes this algorithm section on the stored data and, depending on the outcome, `a` switches into a new state that specifies how to proceed. `a.run()` returns `true` if and only if `a` reached a final state, i. e., if `a` should be popped from the stack.

With this specification of `Algorithm::run()`, the code that triggers both the algorithm execution and the periodical serialization becomes very simple, see Listing 5.1.

The generation of the serialization code that enables the concrete algorithm classes to be serialized through pointers to `Algorithm` strains compilers with suboptimal internal template support (like, e. g., GCC 3.4). The instantiation and compilation of the implementations of the reduction algorithms requires considerable compiler resources as well. We therefore separated the implementations from the algorithm classes as shown in Fig. 5.1. We can thus instantiate algorithm classes and implementations each in their own translation unit and keep the compiler's memory consumption and the total compile time within acceptable limits (up to 350 MByte RAM and approximately 12 minutes, respectively, on our primary test PC with GCC 3.4.3).

The state machine implementation in the `run()` member of the algorithm classes does not need to be sophisticated at all. Any solution more involved than the simple dispatch by a `switch` statement would be overkill and thus hurt the code's accessibility. We see in the excerpt in Listing 5.2 on page 80 that most calls of the individual algorithm parts do not require any arguments; the algorithm implementation objects maintain all necessary context. The only exception are those algorithm parts that may need to call another algorithm. They take the stack onto which they push the new algorithm as parameter.

The sequence diagram in Fig. 5.2 on the next page specifies how LaRed's Sampling Reduction implementation `SRImpl` interacts with the `SRControl`, `SRSampler`, `SRReporter`, and `CSSS` policy objects. For clarity's sake, the diagram omits all function parameters and return values. (The defined policies are depicted in Fig. 5.3 on page 81).

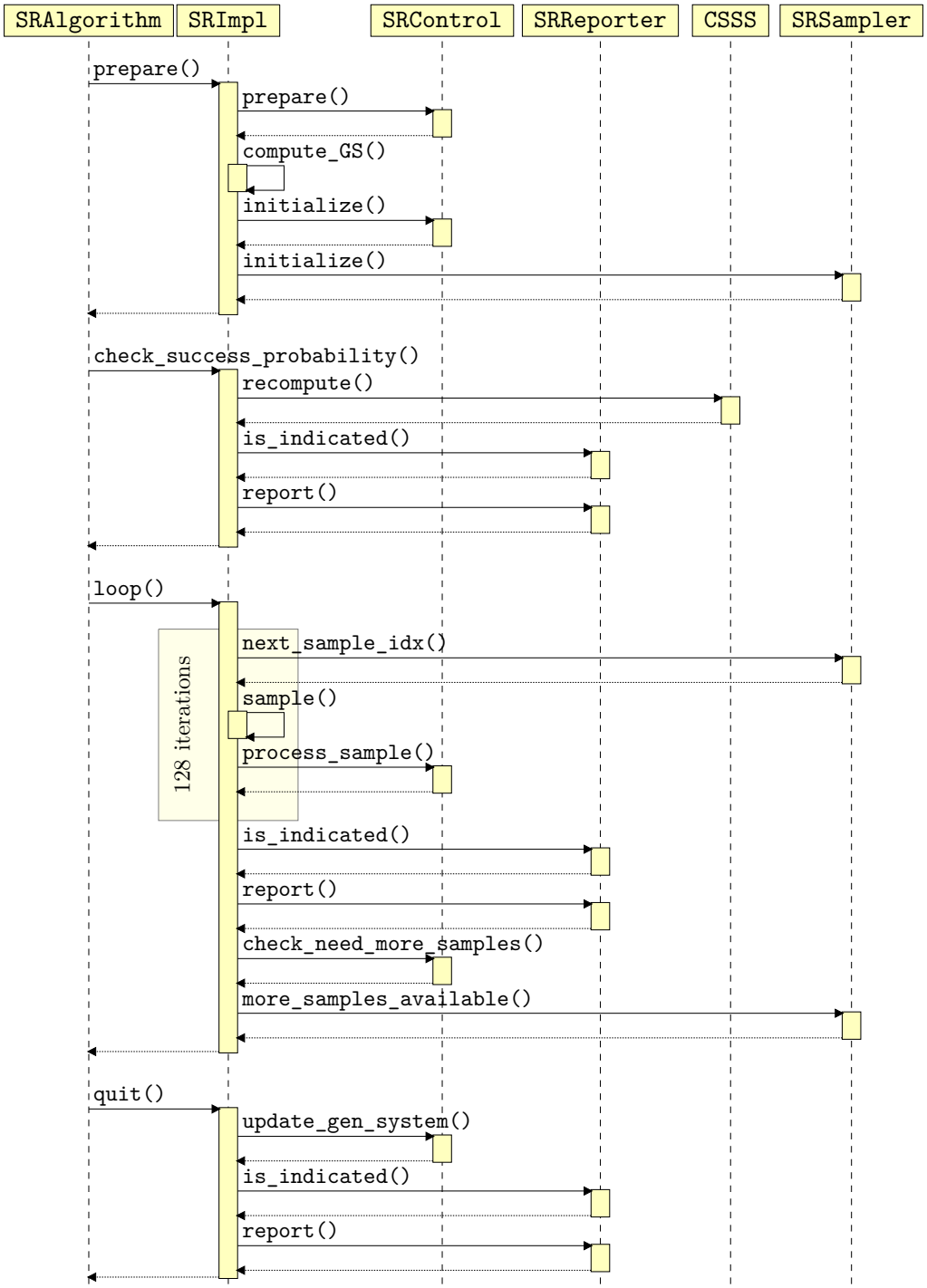


Figure 5.2.: Sampling Reduction Sequence Diagram

```
template<typename IntType, typename FPType>
bool
BKZAlgorithm<IntType, FPType>::
run() {
  switch(this->state) {
    case prepare:
      this->state = this->impl->bkz_prepare();
      break;
    case initial_LLL:
      this->state =
        this->impl->bkz_initial_LLL(this->get_stack());
      break;
    case initial_LLL_postprocess:
      this->state = this->impl->bkz_initial_LLL_postprocess();
      break;
    case enum_prepare:
      this->state = this->impl->bkz_enum_prepare();
      break;

// more cases skipped ...

  }
  return this->state == finished;
}
```

Listing 5.2: Excerpt from `BKZAlgorithm::run()`

The `SRControl` policy assumes a central role. It provides all sampling reduction parameters, including the other policy objects. That way the control policy can enforce that all other policies and parameters are compatible. For instance, `SSRControl` – the control policy that implements SSR – will always return a target set object representing $\{1\}$, the user has no chance to define a different target set which would not make sense for SSR. Similarly, the `ShortProjectionSRControl` object does not accept target set and search space arguments representing T and $V_{\mathbf{c},\mathsf{B}}$, respectively, if there is $1 \leq j \leq \max T$ such that $c_j > 1$ since otherwise the preconditions of CSSS would not hold anymore.

The `SRReporter` object logs status reports to the console or in a log file. Its base class `StatusReporterBase` defines flags that specify whether no log, a periodic log once in a user defined time span, or a log every iteration should be written. Additionally, there's a flag that controls if a log is written when the reduction terminates. During the reduction, `SRImpl` checks with a call to the non-virtual member function `StatusReporterBase::is_indicated()` whether it is supposed to output a log statement via `report()`.

However, the periodic logs require that a reporter object resets the timer in its `StatusReporterBase` subobject whenever `report()` is called, no matter how a user

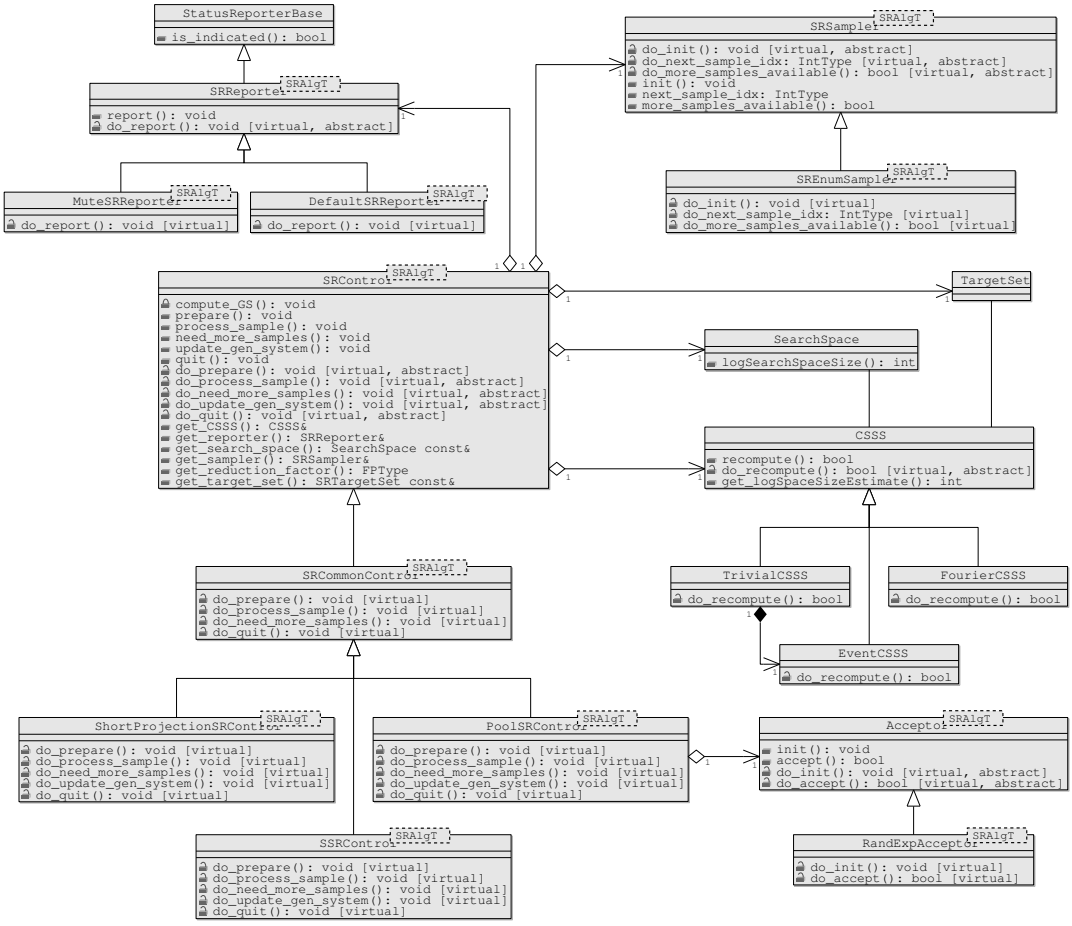


Figure 5.3.: Policies Class Diagram (♦ composition, ◊ aggregation, △ inheritance)

defined `SRReporter` policy overrides the logging function. We therefore applied the Template Method design pattern [GHJV95]. Despite its name, the pattern does not refer to the class and function templates defined in C++. Template Method rather delegates parts of an algorithm to methods that can be overridden by subclasses, but enforces some pre- and postprocessing steps. Here, the `report()` implementation in `SRReporter` resets the timer before it delegates the actual logging to the virtual member function `do_report()`. The `report()` member function is non-virtual whence its invocations from `SRImpl` are statically bound. Even if a user defined logging policy overrides `report()`, LaRed still calls the function defined in `SRReporter`. The only member a user can effectively customize is `do_report()`.

We applied Template Method to LaRed's policy design throughout. We use it, e. g., to `assert` the policies' pre- and postconditions.

Implementations of the `SRControl` interface may factor out some details into their

own policies. So does, e. g., `PoolSRControl`. It defines an `Acceptor` policy that handles the decision whether a sample vector is included in the pool of short vectors. LaRed ships with an `Acceptor` implementation that realizes $\mathcal{R}_{\tau,\alpha}$. Since the random distribution defined by equation (3.29) is somewhat arbitrary, it would be reasonable to implement, say, the acceptor $\mathcal{A}^{(\mathrm{lin})}_{\tau_1,\tau_2}$ subject to

$$\Pr\left[\mathcal{A}^{(\mathrm{lin})}_{\tau_1,\tau_2}(\mathbf{v}) = \mathrm{true}\right] = \begin{cases} 1 & \text{if } \|\mathbf{v}\|^2/\|\mathbf{b}_1\|^2 \leq \tau_1, \\ 0 & \text{if } \|\mathbf{v}\|^2/\|\mathbf{b}_1\|^2 \geq \tau_2, \\ (\|\mathbf{v}\|^2/\|\mathbf{b}_1\|^2 - \tau_1)/\tau_2 & \text{else,} \end{cases}$$

for $0 < \tau_1 < \tau_2$ as well. All that is necessary to make a `ShortPoolSRControl` object use $\mathcal{A}^{(\mathrm{lin})}_{\tau_1,\tau_2}$ is to replace the default `Acceptor` policy object by means of the corresponding setter member function.

The UML-like class diagram in Fig. 5.3 on the page before gives an overview over the policy classes implemented in LaRed. Again, the diagram shows only a selection of the respective class members and omits the function parameters.

The class hierarchies and their interdependencies follow naturally from the design choices described above. There are only few details left worth mentioning: First, the implemented control policies have some overlap; e. g.., all three policies terminate the short vector search if a vector $\mathbf{v}$ was found such that $\|\pi_t(\mathbf{v})\|^2 \leq \gamma\|\hat{\mathbf{b}}_t\|^2$ for any t in the target set T. To avoid code duplication, we defined the intermediate class `SRCommonControl` that implements the behaviour shared by all LaRed control policies. `SRCommonControl` does not override all abstract member functions in `SRControl`, though, whence it is not possible to allocate an object of type `SRCommonControl`.

Second, there is no reason for `CSSS` to store either the target set or the search space. The policy objects potentially need this data only in `do_recompute()`, which determines a new estimate for the search's success probability. This member function is called once per sampling reduction iteration, and both the search space and the target set may be modified by the user between iterations. Therefore, both are passed to `CSSS::recompute()` as parameters. Fig. 5.3 reflects this by showing an association only between `SearchSpace` and `TargetSet` on the one hand and `CSSS` on the other hand, the most loose coupling there is.

Finally, Fig. 5.3 exhibits a somewhat perplexing composition: The implementation of `TrivialCSSS` contains an `EventCSSS` object. The reason is that we want to be able to document the estimated success probability even if we enumerate the search space independent from the success probability. So `TrivialCSSS::do_recompute()` forwards to the `recompute()` member of the contained `EventCSSS` object, but always returns `true`.

5.2. laredpy

When performing experiments with the sampling reduction algorithms implemented in LaRed, we often had to change parameters and even policies between iterations,

based on the result of the previous iteration. We therefore needed a user interface that allows to inspect the reduction results, modify arbitrary policy parameters, and store the results in a format on disk that is suited for further postprocessing. Implementing such an interface – console based or using some GUI toolkit – is tedious, error prone, and often requires platform specific code.

A common workaround is to write an application that takes the information which policies to apply as well as all necessary parameters from the command line. Experiments are then run either interactively from the command shell or automated through shell scripts.

This approach suffers from disadvantages that made it impracticable for our purpose: The frequent setup and tear down of processes is expensive; in particular because the lattice basis must, for each iteration, pass twice through the I/O system – the process has to read it in the beginning and then has to output the new generating system at the end. The parsing of command lines that need to be validated, interpreted, and checked for the mutual compatibility of the given options is complicated as well. So we'd experience a significant performance loss when the iterations at the beginning of sampling reduction often take seconds only. Furthermore, the long lists of command line arguments are cumbersome to type. But, most importantly, command shells were not designed to handle arbitrary data. They typically support rudimentary string handling and, in some cases, elementary integer arithmetic; they are certainly not suited for operations on large floating point arrays etc. That is necessary, though, if we want to evaluate the previous iteration's result and automatically decide the parameters for the next iteration.

We therefore needed a higher level scripting environment as provided by commercial applications like Maple or Matlab or by scripting languages like Perl, Python, or Ruby. The former offer good support for the mathematical evaluation and visualization of data, but lack somewhat when it comes to, e. g., file handling and interaction with user defined native code. We therefore decided to use a general purpose scripting language and settled on Python [Pyt05] because of its clean syntax, the huge repository of available libraries, and, last but not least, its support for native extensions.

5.2.1. The LaRed Binding to Python

Python defines a C API that can be used to implement so called *extension modules*, i. e., software packages implemented in another programming language, typically a compiled language like C or C++. In fact, the better part of Python's standard library is implemented as extension modules. Except in rare corner cases, pure Python modules behave the same as extension modules.

For this to work, one has to register all exported user defined data types with the Python interpreter and wrap all functions in some interface code. Python is a dynamically typed language; in principal, one can pass arbitrary argument lists to functions. The wrapper code around the native functions therefore needs to check and – where reasonable – convert all arguments, allocate new Python objects for the

return values, etc. Manually writing such code is rather involved and tedious.

Fortunately, there are several software packages and code generators that facilitate writing the interface code at a more abstract level. The Boost Python library [Boo04] is specialized in exporting C++ libraries to Python. With Boost.Python, exporting a C++ class reduces to writing few lines to register the type and few lines for each exported function. The library provides the wrapper code needed to guarantee type safety, dispatches calls to the correct function among the set of all possible overloads, maps C++ exceptions to Python exceptions, and so on.

The definition of a Python binding for LaRed was therefore straightforward. We only had to register all classes that are part of LaRed's public NTL interface mentioned in Sect. 5.1.2 as well as their public members with Boost.Python. Alltogether, the C++ code that exports LaRed and the NTL arithmetic classes to Python is less than 10 % the size (in lines of code) of LaRed itself.

Python supports object-oriented programming; in particular, Python has classes and a dispatch mechanism that is similar to the late binding of C++ virtual member functions. If C++ classes with virtual members are properly registered, then the member functions can be overriden by Python instances of this class (or Python classes derived from the C++ class). Boost.Python guarantees that the override takes effect, even if the member function is called from C++ through a base reference. If necessary, then Boost.Python redirects the call back to the Python interpreter. This happens completely transparent for the C++ code. We thus have a mechanism to modify and create sampling policies at runtime. This is a byproduct of the binding via Boost.Python and does not require extra code.

The separation of Sampling Reduction in independent policies helped with keeping LaRed's source base manageable since we no longer need NM functions to cover all combinations of the $N + M$ available implementations of any two policies. It also means, though, that the user has to construct all the policy objects and assemble the actual reduction algorithm. This is not a considerable inconvenience when LaRed is called from C++ since there the code is not rewritten often. But it is a hurdle in an interactive setting where the user has to type the corresponding code every time a session is started. laredpy therefore defines convenience functions for the most frequently used policy combinations that set up the policy objects all at once. These convenience functions provide reasonable defaults for most arguments. Due to a Python feature called *keyword arguments*, the user needs to provide – in no particular order – those arguments only that actually differ from the defaults. (This is different from, e. g., C++, where only trailing default arguments may be omitted.)

5.2.2. Evaluating Reduction Algorithms with laredpy

For the evaluation of the proposed reduction algorithms, we had to run numerous test series and log both the results as well as information about the reduction in a way that facilitates further postprocessing. Due to the amount of data involved – the logfiles alone occupied more than 400 MByte of diskspace – a systematic approach was necessary. The following requirements on the experiment harness transpired

early on:

a) The logs must clearly identify individual experiments.
b) The data of a concluded experiment must not be modified anymore; in particular, the system has to prevent that log files are accidentally overwritten in a subsequent experiment.
c) The data preserved has to be comprehensive. It must be possible to reconstruct all calls of the reduction functions including all policies and parameters. Storing all intermediate generating systems is impractical for the excessive disk space consumption, but at least characteristics like the norm of both the base vectors and the corresponding Gram-Schmidt vectors need to be logged.
d) The retrieval and filtering of log data must be straight forward enough to be practical, even in an interactive setting. The data structures used to retrieve log data must aid its evaluation. This is probably illustrated best by a counter-example: It would be most simple to return the logfile's content as a flat string. However, that puts the burden of parsing the string on the user and therefore hinders its processing, visualization, and interpretation.

Implementing such an experiment harness is a task where we could bring to bear the advantages of Python. In particular, Python's flexible data structures in combination with its support for runtime class interface manipulations and its interpreted nature were crucial in meeting above requirements with surprisingly little code.

We adopted the rule that for each experiment there should be a separate directory in the file system where to write all relevant data. Each experiment is uniquely identified by its log directory. At the same time, the user is free to choose the labels of the experiments in any way that can be mapped to file system paths. Expressive labels help to browse large sets of experiments. For instance, we kept the experiments in a directory tree where each level specified the type of lattice (Micciancio, subset-sum, NTRU, etc.), a serial number of the particular lattice, the LLL-type algorithm used in the reduction and its parameter, the general Sampling Reduction strategy, and so on. This type of hierarchy is not imperative, though. The user can settle on any suitable directory structure.

A reduction can only be reconstructed if the input basis is available. The experiment harness of laredpy therefore requires that any input basis is read from a file on disk; it cannot be set up to log the reduction of bases generated on the fly. (The reduction algorithms do accept such bases, it is only the experiment harness that enforces this rule for replicability's sake.)

An experiment is started by a call to `startExperiment(source, logdir)`. The argument `source` names the file where the input basis is stored, `logdir` specifies the directory associated with the new experiment. `startExperiment` creates this directory if necessary; if it already exists, then the function checks that it is empty. That way we can guarantee that laredpy never overwrites a previous experiment. Any attempt to do so results in an immediate exception.

`startExperiment` returns a nested Python dictionary – referred to as `expdict` in the following – that stores all information about the experiment, including the

logged data. A dictionary is an associative array: Its elements are accessed with the familiar square bracket subscription syntax, but the indices or *keys* do not need to be consecutive integers. laredpy uses strings as keys throughout.

`expdict` holds the full path to the source file and the name of the host where this path is valid and the experiment is run. This information is also written to the file named `source`. The current basis system of the lattice is accessible as `expdict["basis"]`. The basis is written to the file named `result` by calling `finishExperiment(expdict)`.

The most crucial data for the evaluation of an experiment is kept in the dictionary entry `expdict["log"]`. This entry references a Python list that holds for each reduction performed – LLL-type or Sampling Reduction – another dictionary. These nested dictionaries contain the actual log records.

If `expdict` is passed to a laredpy reduction function, then the function adds a log entry to `expdict["log"]`. This entry always contains the key `"algorithm"` with the possible values `"LLL"`, `"BKZ"`, or `"SR"` and and the key `"runtime"` that stores the user CPU time (in seconds) spent on this reduction. All other entries depend on the algorithm and the selected policies.

laredpy takes advantage of a Python feature to keep the generation of the log entries simple: One can query and manipulate the set of attributes and methods of already constructed objects. In particular, one can insert code in an existing object; so, whenever the user creates an instance of one of the policy classes defined in LaRed, then laredpy inserts a readonly property `logentry` into the policy object. The value of this property is a dictionary that holds all key-value-pairs that describe the log info pertaining to this policy object – policy parameters set by the user as well as data computed by the policy object like, e. g., the search space size estimated by a `CSSS` object. This dictionary is constructed whenever `logentry` is accessed. Therefore, laredpy's reduction functions only need to merge the values of the `logentry` properties of all all policy objects into the function's log record and append the resulting dictionary to `expdict["log"]`. In result, `expdict` holds all data necessary to reproduce the reduction and the available information on the effect of the reductions.

Each time a log entry is added to `expdict["log"]`, it is also appended in a "pretty-printed" textual representation to the file `logfile`. The chosen format is in fact a valid Python expression that evaluates to the log record. Therefore, `expdict["log"]` can be efficiently reconstructed by passing the content of `logfile`, enclosed by the strings `"["` and `"]"`, to Python's `eval` function. This is implemented in laredpy by the function `readLog(expdir)` that expects the path to the experiments log directory and returns the corresponding `expdict` structure as above.

The nested dictionary structure of the log data in combination with Python's support for anonymous functions and so called list comprehension – a concise mechanism to build new lists by filtering and manipulating the elements of existing ones – is very convenient to process the log data. For example,

```
[ rec["samples"] for rec in expdict["log"] if "samples" in rec ]
```

returns a list with the number of samples generated in each Sampling Reduction invocation in the course of the experiment. It is therefore easy, both in an interactive setting and in a script, to apply some statistical function on the log data or pass it on to a Python library for data visualization like, e. g., gnuplot's Python interface.

Chapter 6.

Cryptographic Applications

Lattice theory has many applications in, e. g., integer programming [Len83] and communication theory [AEVZ02]. Arguably, the majority of applications of computational lattice theory stems from cryptography, though.

We study in this chapter the impact of Sampling Reduction on some cryptographic applications. Each of these applications operates on its own class of lattices. In the cryptanalysis of Micciancio's variant of the GGH cryptosystem [GGH97, Mic01b], we encounter "random" lattices like the ones we used to demonstrate the behavior of Sampling Reduction in Chapters 2 and 3. The reduction of knapsack lattice bases was used to break several subset-sum based cryptosystems [Odl90, BO91]. If applied directly, this approach can break cryptosystems based on low density subset-sum problems only. However, a more general analysis [OT03] of the results in [LO85, CJL+92] indicates that an improved lattice basis reduction can break high density knapsacks like the ones constructed in the cryptosystem proposed by Okamoto, Tanaka, and Uchiyama [OTU00] as well. From a practical point of view, the NTRU cryptosystem [HPS98, HHGP+03] is currently the most important lattice based cryptosystem. Its security relies on the hardness of finding short vectors in certain modular convolution lattices.

6.1. Micciancio's GGH Variant

In 1997, Goldreich, Goldwasser, and Halevi [GGH97] proposed a cryptosystem – commonly referred to as GGH – that is based on the hardness of the Closest Vector Problem (CVP). The CVP is NP-hard [EB81]; but if the point $\mathbf{x} \in \mathbb{Z}^d$ is not too far away from the lattice L, then the closest lattice point can indeed be efficiently computed by, e. g., Babai's nearest plane algorithm [Bab86]. The admissible distance $\operatorname{dist}(\mathbf{x}, L) = \min\{\, \|\mathbf{v} - \mathbf{x}\| \mid \mathbf{v} \in L \,\}$ depends on the available basis of L. In the GGH system, both the private key K and the public key H are bases of a lattice L, but K allows to solve CVP instances $(\mathbf{x}, L)$ for points x much further away from L than H does. The GGH system assumes it is hard to recover a basis B from H that admits a distance comparable to K.

The way GGH generated the keys and encoded the messages in the point $\mathbf{x} \in \mathbb{Z}^n$ turned out to be flawed; the system was broken in 1999 by Nguyen [Ngu99]. However, the system can be seen as a blueprint for encryption and signature primitives based

Algorithm 13 ReduceModBasis

Input: • $\mathbf{v} = (v_1, \ldots, v_n)^t \in \mathbb{Z}^n$
• lattice basis $\mathsf{H} = [\mathbf{h}_1, \ldots, \mathbf{h}_n] = (h_{i,j}) \in \mathbb{Z}^{n \times n}$ in Hermite normal form
Output: $\mathbf{v}' \in \mathbb{Z}^n$ such that $\mathbf{v}' - \mathbf{v} \in L(\mathsf{H})$ and $\mathbf{v}' = \mathsf{H}\mathbf{x}'$ for some $\mathbf{x}' \in [0, 1)^n$

```
for j from n downto 1 do
    v ← v − ⌊v_j / h_{j,j}⌋ h_j
end for
return v
```

on CVP. Micciancio [Mic01b] proposed in 2001 a variant of GGH that uses a different encoding of the message in $\mathbb{Z}^n$ and that guarantees that the public key does not leak any information about the private key that an attacker could not efficiently compute from any other basis as well. As a side effect, the space asymptotically occupied by both public keys and ciphertexts dropped in Micciancio's scheme by one power of n compared to the original GGH proposal.

Micciancio's proposal describes an encryption scheme, but Goldreich, Goldwasser, and Halevi already outlined how such an CVP based encryption primitive can be turned into a signature primitive [GGH97].

6.1.1. Micciancio's Cryptosystem

Micciancio's system utilizes normalizations of both lattice bases and ciphertexts. Since an attacker can compute these normalizations anyway, they do not incur any weaknesses.

A lattice basis $\mathsf{H} = (h_{i,j}) \in \mathbb{Z}^{n \times n}$ is said to be in *Hermite normal form* if and only if H is upper triangular and $0 \leq h_{i,j} < h_{i,i}$ for all $1 \leq i < j \leq n$. It is well known from linear algebra that, for any lattice basis $\mathsf{B} \in \mathbb{Z}^{n \times n}$, there is a unique basis transformation $\mathsf{U} \in \mathrm{GL}_n(\mathbb{Z})$ such that $\mathsf{H} = \mathsf{BU}$ is in Hermite normal form. In other words, every lattice has exactly one basis H in Hermite normal form. Given any basis, there are polynomial time algorithms to compute H, cf. [Coh96], for instance.

The second normalization is basis dependent: Let $\mathsf{B} \in \mathbb{Z}^{n \times n}$ a lattice basis and $\mathbf{v} \in \mathbb{Z}^n$. We replace $\mathbf{v}$ by the unique representative $\mathbf{v}' \in \mathbb{Z}^n$ of $\mathbf{v} + L(\mathsf{B})$ that lies within the parallelepiped spanned by the columns of B. In principle, we can efficiently compute $\mathbf{v}'$ for any $\mathbf{v}$ and basis B; but it is particularly simple if B is in Hermite normal form, see Alg. 13.

The encryption primitive proposed by Micciancio is as follows. The security parameter n is the dimension of the lattice used to hide the message.

Key Generation. Choose a random lattice basis $\mathsf{B} \in_R \{-n, \ldots, n\}^{n \times n}$, $\det \mathsf{B} \neq 0$. The private key is its LLL reduction $\mathsf{K} = \mathsf{LLL}(\mathsf{B})$, the corresponding public key is (H, ρ) where $\mathsf{H} = \mathrm{HNF}(\mathsf{B})$ is the Hermite normal form of B and $\rho > 0$ is the maximum distance such that knowledge of K allows solving all CVP instances $(\mathbf{x}, L(\mathsf{H}))$ with $\mathrm{dist}(\mathbf{x}, L(\mathsf{H})) \leq \rho$. (For instance, Babai's algorithm solves the CVP if all Gram-Schmidt vectors of K are longer than 2ρ.)

Algorithm 14 Babai

Input:
- lattice basis $\mathsf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{n \times n}$ with Gram-Schmidt decomposition $\mathsf{B} = \hat{\mathsf{B}}\mathsf{R}$
- squared lengths of Gram-Schmidt vectors $\mathfrak{b} = (\|\hat{\mathbf{b}}_1\|^2, \ldots, \|\hat{\mathbf{b}}_n\|^2)$
- vector $\mathbf{c} \in \mathbb{Z}^n$ such that $\mathrm{dist}(\mathbf{c}, L(\mathsf{B})) \leq \min\{\, \|\hat{\mathbf{b}}_j\|/2 \mid j = 1, \ldots, n \,\}$

Output: $\mathbf{v} \in L(\mathsf{B})$ next to $\mathbf{c}$

```
v ← 0
for j from n downto 1 do
    x_j ← ⌈⟨c, b̂_j⟩/‖b̂_j‖²⌋          /* round to closest integer */
    c ← c − x_j b_j
    v ← v + x_j b_j
end for
return v
```

Encryption. The sender encodes the message by some agreed upon enumeration method as a vector $\mathbf{m} \in \mathbb{Z}^n$ subject to $\|\mathbf{m}\| \leq \rho$. The ciphertext is then computed by $\mathbf{c} = \mathsf{ReduceModBasis}(\mathbf{m}, \mathsf{H})$.

Decryption. The receiver computes $\mathbf{v} \in L(\mathsf{K})$ such that $\|\mathbf{v} - \mathbf{c}\| \leq \rho$ using, e. g., Babai's nearest plane algorithm. The message is then recovered as $\mathbf{m} = \mathbf{c} - \mathbf{v}$.

This method results in a strikingly simple and elegant encryption scheme, but, unfortunately, it is impractical. If the dimension n of a Micciancio key is less than 800, then the estimated effort to break this instance of the scheme is less than the threshold stipulated for secure cryptosystems by Lenstra and Verheul [LV01] – even if the only lattice basis reduction available to an attacker is BKZ [Lud02].

In dimension 800, the keys H and K occupy about 1 MByte and 690 KByte, respectively. The key generation is prohibitively slow: In our experiments on a Sun Blade 100 machine with a 500 MHz UltraSparc IIE processor, the computation of a public key for $n = 800$, using the implementation of Kannan's HNF algorithm in LiDIA [LG04], took about 46 hours.

The encryption took about 0.29 seconds per message on said machine. This is still orders of magnitude slower than, say, RSA, but it might be possible to significantly speed it up by a less naïve implementation. However, the decryption poses a much more serious problem: Babai's algorithm as given in Alg. 14 requires exact arithmetic. Since $\|\mathbf{c}\|$ is typically huge, we had to compute the Gram-Schmidt coefficient matrix with a precision of several thousand bits to successfully decrypt messages with this algorithm. The required space to hold the Gram-Schmidt coefficients – far more than 100 MByte – renders this impractical, though.

We therefore followed a heuristic approach: Let $\mathbf{v}'$ be the lattice point by a floating point implementation of Alg. 14. If $\|\mathbf{c} - \mathbf{v}'\| \leq \rho$, then $\mathbf{v}'$ is the closest point to $\mathbf{c}$. Otherwise, we expect $\|\mathbf{c} - \mathbf{v}'\| < \|\mathbf{c}\|$, whence we apply Alg. 14 again to $\mathbf{c}' = \mathbf{c} - \mathbf{v}$. With this method, we could reliably decrypt messages, but decryption took more than one hour per message in dimension 800.

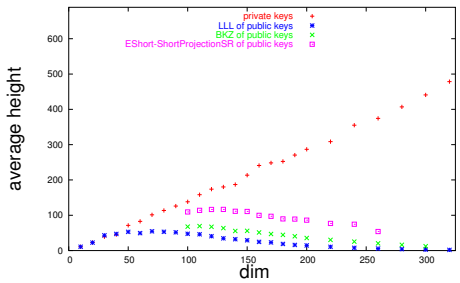


Figure 6.1.: Average height of private and reduced public Micciancio keys ($\delta = 0.99$, $\beta = 5$, $u_{\max} = 22$, $\mathsf{CSSS}_{\mathsf{triv}}$)

In summary, Micciancio's scheme is not appropriate for practical applications. However, variants in special classes of lattices that allow a more compact representation of the bases may turn out to be practical. For example, in the NTRUSign system, each public key of size $O(n)$ is associated with a $2n$-dimensional lattice. NTRUSign uses the ability to solve an approximate CVP in this lattice as proof of possession of the private key, similar to the signature variant of Micciancio's scheme.

For our purposes, the perhaps most important feature of Micciancio's scheme is that it does not impose any particular structure on the lattices used. For the lack of a natural distribution on all lattices, the lattices used in Micciancio's scheme, compared to the other classes of lattices considered in this chapter, come arguably closest to the intuitive notion of random lattices. Micciancio's scheme therefore serves as an important benchmark for the performance of lattice basis reduction algorithms.

6.1.2. Lattice Basis Reduction Attacks

Recovering a message in the Micciancio scheme is equivalent to (exactly) solving some CVP instance. Agrell et al. [AEVZ02] survey the known deterministic CVP algorithms; their experiments show exponential behavior for all of them. On the other hand, the nearest plane algorithm (Alg. 14 on the preceding page) by Babai – which, in general, solves an approximate version of the CVP only – enables the holder of a private Micciancio key K to find in polynomial time the lattice point $\mathbf{v}$ closest to the ciphertext $\mathbf{c}$ provided $\|\mathbf{c} - \mathbf{v}\| \leq \min\{\, \|\hat{\mathbf{k}}_j\| \mid j = 1, \ldots, n \,\}$.

An attacker can therefore either try to reduce the public key such that the height of the reduced basis becomes large enough to make decrypting feasible. Or the attacker targets a single ciphertext and constructs a new lattice problem that is likely to be easier to solve.

Fig. 6.1 shows that the former approach is not feasible with LLL-type reductions for large n. The height of the reduced public keys becomes soon negligible, even with BKZ.

As discussed in Chapter 3, SSR and PoolSR have not much effect on the height of

the reduced bases. Hence, these algorithms are not a reasonable alternative for an attacker. ShortProjectionSR, in contrast, does significantly affect the length of the last Gram-Schmidt vectors. The 4th graph in Fig. 6.1 therefore shows the average height after application of ShortProjectionSR to the already BKZ reduced ($\beta = 5$) public keys with target index set $\{1, \ldots, 10\}$, i. e., the target set that corresponds to Schnorr's EShort. It is apparent that the height of the result basis the attacker obtains is much larger, but for increasing n the height again declines. The difference between the height of a such reduced public key at dimension n and the corresponding private key is roughly the same as the difference between a BKZ reduced public key and its corresponding private key at dimension $n - 50$. In other words, Sampling Reduction increases the dimension up to which this attack is viable by about 50.

The second approach that targets a single ciphertext only is realized by the so called embedding attack [GGH97]. For this attack, we consider the lattice L' generated by the block matrix

$$\mathsf{A}' = \begin{pmatrix} \mathsf{A} & \mathbf{c} \\ \mathbf{0} & 1 \end{pmatrix} \in \mathbb{Z}^{(n+1)\times(n+1)} \tag{6.1}$$

where A is the basis the attacker obtained by reducing the public key H and $\mathbf{c}$ is the attacked ciphertext. Let $\mathbf{v}$ be the lattice point in L' closest to $\mathbf{c}$. Clearly, $\mathbf{x}' = (\mathbf{c} - \mathbf{v}, 1)^t \in L(\mathsf{A}')$ and $\|\mathbf{x}'\|^2 \leq \rho^2 + 1$. Thus, heuristically, $\mathbf{x}'$ is a short or even shortest vector in L'. The closer $\mathbf{c}$ to $\mathbf{v}$, the larger is the likely gap between $\lambda_1(L')$ and $\lambda_2(L')$ and the better are the chances to recover $\mathbf{x}'$ by lattice basis reduction.

The embedding attack is surprisingly powerful. Table 6.1 on the following page shows the ratio and average runtime of successful attacks with LLL depending on the lattice dimension and on the distance of the ciphertext to the lattice. In key dimension 150 and lower, we did not observe any failures. $n = 190$ was the minimum dimension where an embedding attack with LLL consistently failed for ciphertexts far enough from the lattice. Successful attacks terminated within minutes, at most.

As expected, BKZ reduction pushes the dimension where the embedding attack fails (Table 6.2 on page 95). With $\beta = 5$, it is only in $n = 220$ and higher that we observed any failures. However, the runtime of successful attacks varied considerably and is, on average, higher by a factor about 20.

Since we are only interested in a single short vector, SSR is the most appropriate non-interactive Sampling Reduction algorithm. Table 6.3(a) on page 96 shows that, if LLL alone fails, then SSR used in conjunction with LLL is in fact a viable alternative to BKZ that runs much faster. Unfortunately, SSR does not succeed as reliably as BKZ; it failed to break most ciphertexts in dimension 240. But we have to point out that these experiments were run with the rather small search space size $u_{\max} = 20$ and this search space was rarely exhausted because $\mathsf{CSSS}_{\mathsf{event}}$ soon indicated a small success probability. In fact, in most experiments only up to $\approx 2^{15}$ vectors were sampled. We can therefore expect to break more instances if we accept larger search spaces and consequently longer runtimes.

We have a similar result for our experiments with SSR combined with BKZ, $\beta = 5$, cf. Table 6.3(a). SSR could break some ciphertexts that withstood BKZ alone. The

success ratio avg. runtime	dist$(\mathbf{c}, L) =$ 0.10ρ	0.20ρ	0.30ρ	0.40ρ	0.50ρ	0.60ρ	0.70ρ	0.80ρ	0.90ρ	1.00ρ
dim = 91	45 / 45 0.0	45 / 45 0.0	45 / 45 0.0	45 / 45 0.0	45 / 45 0.0	45 / 45 0.0	45 / 45 0.0	45 / 45 0.0	45 / 45 0.0	45 / 45 0.0
dim = 161	50 / 50 5.5	50 / 50 6.8	50 / 50 6.5	50 / 50 7.4	50 / 50 6.6	50 / 50 6.0	50 / 50 5.3	45 / 50 6.6	42 / 50 7.1	39 / 50 8.1
dim = 171	50 / 50 6.7	50 / 50 6.6	50 / 50 6.9	50 / 50 7.9	50 / 50 6.7	47 / 50 6.8	46 / 50 7.4	39 / 50 8.0	36 / 50 8.3	31 / 50 6.7
dim = 181	50 / 50 8.2	50 / 50 8.7	50 / 50 10.6	50 / 50 11.2	50 / 50 9.6	45 / 50 8.4	37 / 50 10.1	16 / 50 5.8	19 / 50 6.5	15 / 50 12.7
dim = 191	50 / 50 10.9	50 / 50 14.4	50 / 50 12.8	50 / 50 10.0	48 / 50 14.6	43 / 50 11.1	30 / 50 10.8	14 / 50 6.3	3 / 50 3.0	0 / 50
dim = 201	25 / 25 14.0	25 / 25 11.6	25 / 25 14.2	25 / 25 15.4	20 / 25 14.1	12 / 25 10.5	6 / 25 16.8	0 / 25	0 / 25	0 / 25
dim = 221	25 / 25 19.7	25 / 25 24.1	24 / 25 15.8	14 / 25 17.5	10 / 25 11.1	0 / 25	0 / 25	0 / 25	0 / 25	0 / 25
dim = 241	25 / 25 43.8	24 / 25 34.9	6 / 25 10.0	0 / 25	0 / 25	0 / 25	0 / 25	0 / 25	0 / 25	0 / 25
dim = 261	15 / 15 91.9	12 / 15 41.4	2 / 15 34.0	0 / 15	0 / 15	0 / 15	0 / 15	0 / 15	0 / 13	0 / 10
dim = 281	5 / 5 56.2	0 / 5	0 / 5	0 / 5	0 / 5	0 / 5	0 / 5	0 / 5	0 / 5	0 / 5
dim = 301	4 / 5 66.8	0 / 5	0 / 5	0 / 5	0 / 5	0 / 5	0 / 5	0 / 5	0 / 5	0 / 5

Table 6.1.: Embedding attack on Micciancio messages with LLL ($\delta = 0.99$)

success ratio avg. runtime	dist($\mathbf{c}, L$) =									
	0.10ρ	0.20ρ	0.30ρ	0.40ρ	0.50ρ	0.60ρ	0.70ρ	0.80ρ	0.90ρ	1.00ρ
dim = 161								2 / 2 102.0	3 / 3 96.3	4 / 4 98.5
dim = 171						3 / 3 137.0	4 / 4 138.5	11 / 11 112.0	14 / 14 129.4	19 / 19 115.5
dim = 181						2 / 2 165.5	5 / 5 174.8	12 / 12 171.5	15 / 15 186.3	13 / 13 175.4
dim = 191						2 / 2 237.0	2 / 2 248.0	8 / 8 234.8	9 / 9 231.4	10 / 10 238.1
dim = 201					1 / 1 290.0	4 / 4 292.8	5 / 5 310.0	5 / 5 305.8	5 / 5 295.4	5 / 5 309.0
dim = 221				3 / 3 483.0	2 / 2 512.5	5 / 5 506.6	5 / 5 487.4	5 / 5 486.8	5 / 5 547.0	4 / 5 494.8
dim = 241		1 / 1 778.0	5 / 5 799.0	5 / 5 807.8	5 / 5 791.4	5 / 5 810.4	5 / 5 792.4	5 / 5 830.6	3 / 5 886.0	3 / 5 830.7
dim = 261	2 / 2 110.5	3 / 3 421.3	5 / 5 729.2	5 / 5 718.0	5 / 5 741.2	5 / 5 803.0	5 / 5 738.2	2 / 5 147.5	1 / 5 47.0	1 / 5 200.0
dim = 281	4 / 4 231.5	5 / 5 518.2	5 / 5 610.2	5 / 5 531.2	4 / 5 233.2	2 / 5 96.5	1 / 5 87.0	0 / 5	0 / 5	0 / 5
dim = 301	4 / 4 237.5	4 / 4 209.0	4 / 4 220.8	3 / 4 245.7	1 / 4 247.0	0 / 4	0 / 4	0 / 4	0 / 4	0 / 3

Table 6.2.: Embedding attack on Micciancio messages with BKZ ($\beta = 5$)

success ratio avg. runtime	dist$(\mathbf{c}, L) =$ 0.10ρ	0.20ρ	0.30ρ	0.40ρ	0.50ρ	0.60ρ	0.70ρ	0.80ρ	0.90ρ	1.00ρ
dim = 161								3 / 9 7.0	8 / 11 10.5	11 / 15 12.0
dim = 171						3 / 6 15.7	4 / 8 7.0	11 / 22 8.5	9 / 28 14.0	10 / 38 15.1
dim = 181						5 / 7 10.4	10 / 18 17.1	30 / 41 17.7	16 / 39 17.4	21 / 44 25.4
dim = 191					2 / 2 17.0	7 / 7 25.1	17 / 20 22.6	26 / 36 26.1	31 / 47 26.5	22 / 50 26.5
dim = 201					5 / 5 20.8	13 / 13 29.8	12 / 19 31.9	13 / 25 25.3	8 / 25 25.2	5 / 25 26.6
dim = 221			1 / 1 25.0	8 / 11 36.9	13 / 15 41.9	5 / 25 51.0	8 / 25 63.8	0 / 25	0 / 25	0 / 25
dim = 241			7 / 8 55.0	1 / 10 112.0	0 / 10	0 / 10	0 / 10	0 / 10	0 / 10	0 / 7

(a) intermittent LLL reductions ($u_{\max} = 20$)

success ratio avg. runtime	dist$(\mathbf{c}, L) =$ 0.10ρ	0.20ρ	0.30ρ	0.40ρ	0.50ρ	0.60ρ	0.70ρ	0.80ρ	0.90ρ	1.00ρ
dim = 221										1 / 1 555.0
dim = 241									2 / 2 983.5	2 / 2 909.0
dim = 261								3 / 3 1510.0	1 / 4 1427.0	2 / 4 804.5
dim = 281					1 / 1 1732.0	1 / 3 229.0	0 / 4	0 / 5	0 / 5	0 / 5
dim = 301				1 / 1 337.0	2 / 3 563.0	2 / 4 831.5	0 / 4	0 / 4	0 / 4	0 / 3

(b) intermittent BKZ reductions ($\beta = 5$, $u_{\max} = 22$)

Table 6.3.: Embedding attack on Micciancio messages with SSR

runtime of these attacks was longer than the successful BKZ attacks in the same dimension by a small factor only.

6.2. NTRU

The NTRU cryptosystem was originally proposed by Hoffstein, Pipher, and Silverman at the rump session of Crypto '96. It was pointed out early on that breaking NTRU instances is equivalent to finding very short vectors or solving CVPs in a very particular class of lattices. In fact, Coppersmith and Shamir [CS97] published a thorough analysis of the relation between NTRU and lattice problems even before NTRU was formally published [HPS98].

NTRU appeals for two reasons: First, NTRU operates on elements of the factor ring $R := \mathbb{Z}[X]/(X^N - 1)$ for some integer N. The multiplication in R is in fact a convolution and therefore allows very efficient implementations. Furthermore, NTRU computations do not require support for large integers which considerably simplifies the implementation of NTRU on small devices. This makes NTRU's performance at least competitive to established cryptosystems like RSA or ECC, if not faster. Second, NTRU is independent from the RSA problem and the discrete logarithm problem. It is therefore a viable alternative in case RSA or ECC should ever be fundamentally broken. This is a particularly attractive argument in view of the discussion about the impact of quantum computing on cryptography; sufficiently large quantum computers can break both RSA and ECC in polynomial time, but there is no known quantum algorithm that breaks NTRU.

On the other hand, the suggested parameters and padding schemes for NTRU had to be revised several times [JJ00, NP02, HGNP+03]. The first NTRU based signature scheme NSS [HPS01] as well as a hastily patched version were almost immediately broken [GJSS01, GS02]. This cost NTRU a lot of confidence. The NTRUSign scheme, proposed in 2002 [HHGP+02], seems to be stronger than NSS; we are not aware of any attacks that fundamentally break it. But there were minor flaws in NTRUSign as well [MYK04] and the key generation and parameter sets are still subject of research [HHGP+05], so it is probably too early to assess the security of NTRUSign.

6.2.1. NTRU Lattices

We sketch in this section the correspondence between instances of the NTRUEncrypt primitive and the NTRU lattices. More details can be found in [CS97] and [HSW03].

Every element of R can be represented by an integral polynomial of degree at most $N-1$. In the NTRU context, the length of such a polynomial $f = \sum_{j=0}^{N-1} f_j X^j$ is measured by the *centered norm*

$$\|f\|_{\mathrm{c}}^2 = \sum_{j=0}^{N-1} (f_j - \overline{f})^2 \qquad \text{with} \qquad \overline{f} = \frac{1}{N} \sum_{j=0}^{N-1} f_j \,.$$

(To be precise, $\|\cdot\|_{\mathrm{c}}$ is a pseudo-norm only since $\|f\|_{\mathrm{c}} = 0$ does not imply $f = 0$. This does not matter for our discussion, though.)

The map $R \to \mathbb{Z}^N : f = \sum_{j=0}^{N-1} f_j X^j \mapsto \mathbf{f} = (f_0, \ldots, f_{N-1})^t$ is an isomorphism of $\mathbb{Z}$-algebras. The multiplication in R corresponds to the convolution product in $\mathbb{Z}^N$, i. e., $R \times R \to R : (f, g) \mapsto f * g$ corresponds to $\mathbb{Z}^N \times \mathbb{Z}^N \to \mathbb{Z}^N : (\mathbf{f}, \mathbf{g}) \mapsto \mathsf{C}_{\mathbf{f}}\mathbf{g}$ where

$$\mathsf{C}_{\mathbf{f}} = \begin{pmatrix} f_0 & f_1 & \cdots & f_{N-1} \\ f_{N-1} & f_0 & \cdots & f_{N-2} \\ \vdots & \vdots & \ddots & \vdots \\ f_1 & f_2 & \cdots & f_0 \end{pmatrix} \in \mathbb{Z}^{N \times N} .$$

NTRUEncrypt has numerous system parameters. Besides the polynomial degree N, one has to fix a small modulus p and a large modulus q such that $pR + qR = R$. The proponents of NTRU discussed the choice $p = X - 2$ [HS00], but the recommended parameter sets suggest $p = 3$. The polynomial sets $\mathcal{L}_f, \mathcal{L}_g \subset R$ describe the private key space; the elements of $\mathcal{L}_f$ need to be invertible both modulo p and modulo q. If $\mathcal{L}_f, \mathcal{L}_g$ are chosen according to the recommended parameter sets, then $\|f\|_{\mathrm{c}}, \|g\|_{\mathrm{c}}$ are public constants independent from $f \in \mathcal{L}_f, g \in \mathcal{L}_g$.

The private key (f, g) is a uniform random element of $\mathcal{L}_f \times \mathcal{L}_g$. The corresponding public key is

$$h = f^{-1} * g \bmod q . \tag{6.2}$$

The lattice $L_{h,q,\alpha}$ is generated by the block matrix

$$\mathsf{B}_{h,q,\alpha} = \begin{pmatrix} \alpha \mathsf{I}_N & 0 \\ \mathsf{C}_{\mathbf{h}} & q\mathsf{I}_N \end{pmatrix} \in \mathbb{R}^{2N \times 2N}$$

where I_N is the N dimensional identity matrix and $\alpha \in \mathbb{R}$ is positive. $L_{h,q,\alpha}$ contains the vector $\mathbf{v}_\alpha := (\alpha\mathbf{f}, \mathbf{g})^t$ since

$$\mathbf{v}_\alpha = \mathsf{B}_{h,q,\alpha} \begin{pmatrix} \mathbf{f} \\ \mathbf{w} \end{pmatrix} \qquad \text{with} \qquad \mathbf{w} = \frac{1}{q}(\mathsf{C}_{\mathbf{h}}\mathbf{f} - \mathbf{g}) \in \mathbb{Z}^N$$

by (6.2). In other words, the private key is essentially a lattice point. Its norm $\|\mathbf{v}_\alpha\|_{\mathrm{c}} := \sqrt{\|\alpha f\|_{\mathrm{c}}^2 + \|g\|_{\mathrm{c}}^2}$ is public knowledge due to the construction of the polynomial spaces $\mathcal{L}_f$ and $\mathcal{L}_g$. It turns out that $\mathbf{v}_\alpha$ is a very short vector in $L_{h,q,\alpha}$; in fact, for suitable α, the norm $\|\mathbf{v}_\alpha\|_{\mathrm{c}}$ is smaller than the estimate $\sigma = \sqrt{Nq\alpha/\pi e}$ for the first minimum $\lambda_1(L_{h,q,\alpha})$ that follows from the Gaussian heuristic. Coppersmith and Shamir [CS97] show that it suffices to find vectors $\mathbf{v}'$ not much longer than $\mathbf{v}_\alpha$, say $4\|\mathbf{v}_\alpha\|_{\mathrm{c}}$, to break NTRU. Thus, the most direct attack on NTRU is to apply lattice basis reduction algorithms to $\mathsf{B}_{h,q,\alpha}$ in the hope the reduction may yield sufficiently short basis vectors.

Empirically, LLL-type reduction algorithms are the more likely to find a short vector $\mathbf{x}$ in a lattice the smaller the ratio between the length of $\mathbf{x}$ and the Gaussian heuristic estimate. An attacker who tries to recover the private key by reduction of $\mathsf{B}_{h,q,\alpha}$ will therefore choose $\alpha = \|g\|_{\mathrm{c}}/\|f\|_{\mathrm{c}}$ as this choice minimizes said ratio. $L_{h,q}^{\mathrm{NT}} := L_{h,q,\|g\|_{\mathrm{c}}/\|f\|_{\mathrm{c}}}$ is the NTRU lattice corresponding to the public key h.

6.2.2. Attacks by Sampling Reduction

We describe the effectiveness of lattice reduction attacks on NTRU by ShortProjection-SR. Our results indicate that the estimated breaking cost for NTRU needs to be reassessed. Such a reassessment needs to take more refined attacks (e. g., zero-forced NTRU lattices [MS01] or meet-in-the-middle attacks [HGSW03]) into consideration as well, whereas we were most interested in how Sampling Reduction performs in the class of NTRU lattices.

For a given NTRU instance with private key (f, g) and public key h set

$$\begin{aligned} a &:= N/q, \\ c &:= \sqrt{4\pi e \|f\|_{\mathrm{c}} \|g\|_{\mathrm{c}}/q} = \sqrt{2N}\|\mathbf{v}\|_{\mathrm{c}}/\sigma \end{aligned}$$

where $\mathbf{v} \in L_{h,q}^{\mathrm{NT}}$ is the vector corresponding to the private key and σ is the Gaussian heuristic estimate for $\lambda_1(L_{h,q}^{\mathrm{NT}})$. Hoffstein, Silverman, and Whyte state in a technical report [HSW03] that, based on their experience, the greater a or c the longer it takes to break NTRU by lattice reduction. They report experimental results for $a = 0.535$ and $c = 1.73$.

We prepared our test cases as outlined by [HSW03]: We chose the elements of $\mathcal{L}_f, \mathcal{L}_g$ as polynomials with exactly d_f, d_g coefficients equal to 1 and all remaining coefficients equal to 0. This implies $\|f\|_{\mathrm{c}}^2 = d_f - d_f^2/N$ and $\|g\|_{\mathrm{c}}^2 = d_g - d_g^2/N$ for all $f \in \mathcal{L}_f, g \in \mathcal{L}_g$. Depending on the security parameter N, we determined q, d_f, and d_g such that a and c were close to but not less than the values used in said technical report and $\alpha \approx 1$. We then chose a random private key (f, g) and computed h according to (6.2). We finally generated the lattice basis

$$\mathsf{B} = s \begin{pmatrix} \tilde{\alpha}\mathsf{I}_N & 0 \\ \mathsf{C}_{\mathbf{h}} & q\mathsf{I}_N \end{pmatrix} \in \mathbb{Z}^{2N \times 2N} \tag{6.3}$$

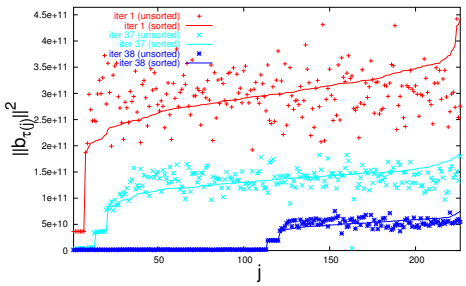
with the scale factor $s = 10^3$ where $\tilde{\alpha} = \lceil s\alpha \rfloor / s$ is α rounded up to three decimal digits.

We reduced such bases for $97 \leq N \leq 113$ with BKZ for successively increasing values of β and additionally with ShortProjectionSR. We exhibit the observed reduction behavior on an example with $N = 113$; it is representative for all experiments.

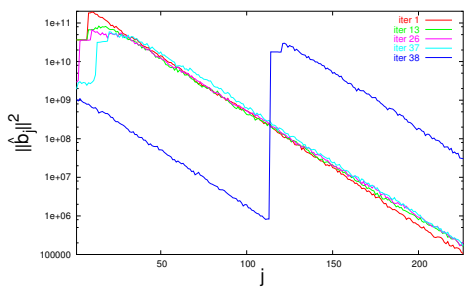
The parameters for the generation of this experiment were $N = 113$, $q = 191$, $d_f = 22$, and $d_g = 21$, which implies $a = 0.592$ and $c = 1.78$. The euclidean norm of the target vector $\mathbf{v}$ was roughly $\|\mathbf{v}\| \approx 6.3s$.

The basis was initially BKZ reduced in 502 seconds with $\delta = 0.99$ and up to $\beta = 5$. This yielded 7 base vectors $\mathbf{b}$ with $\|\mathbf{b}\|^2 = s^2q^2 = 3.6 \times 10^{10}$, all other base vectors were much longer, cf. the top graph in Fig. 6.2(a) on the next page. The BKZ reduced basis did therefore not admit to break this NTRU instance.

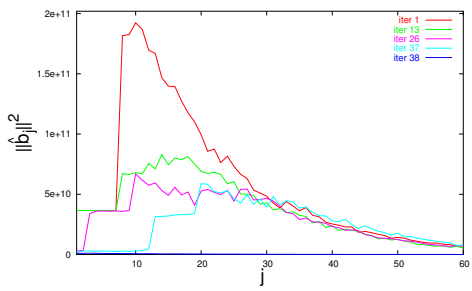
It is also obvious from Fig. 6.2 that the chance to improve the basis with either SSR or PoolSR was negligible since $\|\mathbf{b}_1\|^2 \approx 0.2\|\hat{\mathbf{b}}_8\|^2$. However, ShortProjectionSR is most adequate to reduce $\|\hat{\mathbf{b}}_j\|^2$, $j > 7$. We ran the reduction with reduction



(a) Euclidean base vector lengths.



(b) Logarithmic overview over Gram-Schmidt vector lengths.



(c) Details of Gram-Schmidt vector lengths at front of basis.

Figure 6.2.: NTRU basis under ShortProjectionSR.
($N = 113$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.9$)

factor $\gamma = 0.9$ throughout. The first iterations with target set $T = \{8\}$ returned within seconds. Already after the fourth iteration, the ratio $\|\hat{\mathbf{b}}_8\|^2/\|\hat{\mathbf{b}}_9\|^2$ became such that the chances for a sampled vector to replace $\mathbf{b}_8$ seemed very slim whence we extended the target set to $T = \{8, 9\}$ and in the following iterations gradually to $T = \{8, \ldots, 12\}$. After 17 iterations, the graph of $\|\hat{\mathbf{b}}_j\|^2$ showed a distinct depression around $\hat{\mathbf{b}}_{13}$ whence we had to insert 4 iterations with $T = \{14, \ldots, 18\}$ to reduce the second peak at $\hat{\mathbf{b}}_{18}$. After that we could continue to reduce the vectors near $\mathbf{b}_8$. The BKZ reduction in the 25th iteration resulted for the first time in a vector significantly shorter than sq.

From this point on, every iteration inserted a short vector at the front of the basis and pushed the peak of the Gram-Schmidt vector lengths one slot back whence we chose the target sets $\{9, \ldots, 13\}$, $\{10, \ldots, 14\}$ and so on. The 38th iteration finally broke the instance; it resulted in $30s \leq \|\mathbf{b}_j\| \leq 70s$ for $j \in \{1, \ldots, 113\}$.

Diagrams (a) and (b) in Fig. 6.2 show that the last BKZ reduction breaks the basis down into two blocks of length N; the first one consists of short vectors that essentially reveal the NTRU trapdoor, the second one contains only vectors of length

N	97	100	103	107	109	111	113
BKZ	5	5	5	8	8	8	11
ShortProjectionSR	3	3	3	5	5	5	5

Table 6.4.: Values of β that broke NTRU lattices.

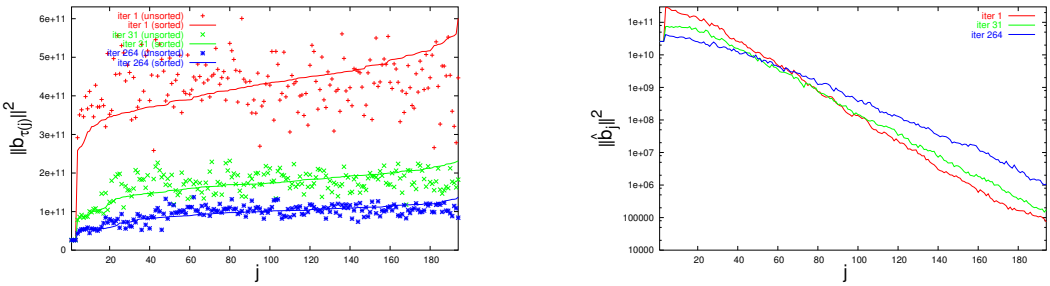
$\approx sq$. The huge jump from $\|\hat{\mathbf{b}}_N\|^2$ to $\|\hat{\mathbf{b}}_{N+1}\|^2$ and $\|\mathbf{b}_{N+1}\|^2 \approx \|\hat{\mathbf{b}}_{N+1}\|^2$ indicate that the two blocks generate almost orthogonal sublattices of $L^{\mathrm{NT}}_{h,q}$. If necessary, few PoolSR iterations with the search space $V_{\mathbf{c},\mathsf{B}}$ (where $\mathbf{c} = (c_1, \ldots, c_{2N})$ and $c_j = 1$ for $j \leq N - 22$, $c_j = 2$ for $N - 22 < j \leq N$, and $c_j = 0$ else) further reduce the base vectors $\mathbf{b}$ in the first block such that $3\|\mathbf{v}\| \leq \|\mathbf{b}\| \leq 5\|\mathbf{v}\|$.

To break the same NTRU lattice with BKZ only, we had to increase the BKZ parameter up to $\beta = 11$. The accumulated runtime for these BKZ reductions was 897 seconds. The total runtime until ShortProjectionSR broke the instance was more than six times longer (5784 seconds). However, 90 % of this time was spent on sampling, only 582 seconds went into the intermittent BKZ reductions. Since sampling can be easily distributed onto several machines with an efficiency near 1 (cf. Section 3.7), an attacker can run a massively distributed variant of ShortProjectionSR on cheap hardware. With approximately 20 nodes, such a reduction of our example basis would probably run not longer than BKZ alone. We are not aware of any BKZ variant that can exploit distributed computing resources in the same way.

There is no rigorous runtime bound for the BKZ algorithm; it is, empirically, exponential in β, though. The values of β required to break NTRU with BKZ increase rapidly with N. E. g., for an experiment with $N = 129$ we already had to reduce the basis with $\beta = 22$ in more than 45 hours. Since the attack with ShortProjectionSR requires smaller values of β (cf. Table 6.4), it seems likely that the gap between the runtime of a BKZ only attack and the runtime spent on BKZ in a ShortProjectionSR attack widens. Then a massively distributed ShortProjectionSR implementation may actually break NTRU instances for large N in less time than BKZ (even though the total number of CPU cycles spent in the attack may be larger).

We conclude this section with some remarks regarding the best strategy for attacking NTRU lattices with ShortProjectionSR. In our experience, we succeeded faster in breaking NTRU lattices if we chose the reduction factor $\gamma = 0.90$ rather than 0.99. The fewer number of iterations outweighed the additional vectors we had to sample in each round.

Second, it proved crucial to restrict the target set to the smallest indices one could conceivably expect to succeed, even though this meant we several times almost exhausted our search space (with $u_{\max} = 22$). If we instead chose a too large target set, then we were likely to run into a situation as exhibited by Fig. 6.3. Here the sampling time was consistently low, but the insertions beyond the peak of the Gram-Schmidt vector lengths caused the length of the Gram-Schmidt vectors $\hat{\mathbf{b}}_j$, $j > 80$,



(a) Euclidean base vector lengths.

(b) Logarithmic overview over Gram-Schmidt vector lengths.

Figure 6.3.: Result of too opportunistic ShortProjectionSR.

to increase. This lowered the probability to find a sufficiently short replacement for $\hat{\mathbf{b}}_4$ to the point where we did not observe any progress for the base vectors near the front of the basis anymore. In the end we had managed to improve the mean of $\|\mathbf{b}_j\|^2$ by a factor 4, but the attack attempt on the NTRU instance had failed. However, we could break this instance with smaller target sets.

6.3. Knapsack Lattices

From the early days of public key cryptography on, cryptosystems based on the *knapsack problem* (also known as *subset sum problem*) were proposed as alternatives to RSA. Even though the knapsack problem is NP-hard in the worst case, all knapsack based cryptosystems were broken. Lattice basis reduction proved to be a key tool in the cryptanalysis of knapsack based cryptosystems [Odl90, NS01].

We explain in the following section the knapsack lattice constructed by Coster et al. [CJL+92]. We then describe in Sect. 6.3.2 how sampling reduction performs on such lattices.

6.3.1. Knapsack Lattice Bases

We define (the search version of) the knapsack problem and describe how Coster et al., based on earlier work by Lagarias and Odlyzko, turn it – for a certain range of parameters – into an SVP.

Definition 18. *The search version of the* knapsack *or* subset sum *problem is to find, for given weights* $a_1, \ldots, a_n \in \mathbb{N}_+$ *and sum* $s \in \mathbb{N}_+$*, a vector* $\mathbf{x} = (x_1, \ldots, x_n)^t \in \{0, 1\}^n$ *such that*

$$s = \sum_{j=1}^{n} x_j a_j \,. \tag{6.4}$$

The quotient

$$d = \frac{n}{\log_2 \max\{\, a_1, \ldots, a_n \,\}} \tag{6.5}$$

is called the density *of the knapsack* $(a_1, \ldots, a_n, s)$.

Given a knapsack problem $(a_1, \ldots, a_n, s)$, Lagarias and Odlyzko [LO85] considered the $n+1$ dimensional lattice generated by

$$\mathsf{B}_{\mathrm{LO}} = \begin{pmatrix} 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 0 \\ Na_1 & Na_2 & \ldots & Na_n & Ns \end{pmatrix} \in \mathbb{Z}^{(n+1)\times(n+1)} \tag{6.6}$$

where $N > \sqrt{n}$. Provided the density of the knapsack is $d < 0.6463$, they could show that, with high probability, the embedded solution vector $(x_1, \ldots, x_n, 0)^t$ of the knapsack is a shortest vector in $L(B_{\mathrm{LO}})$.

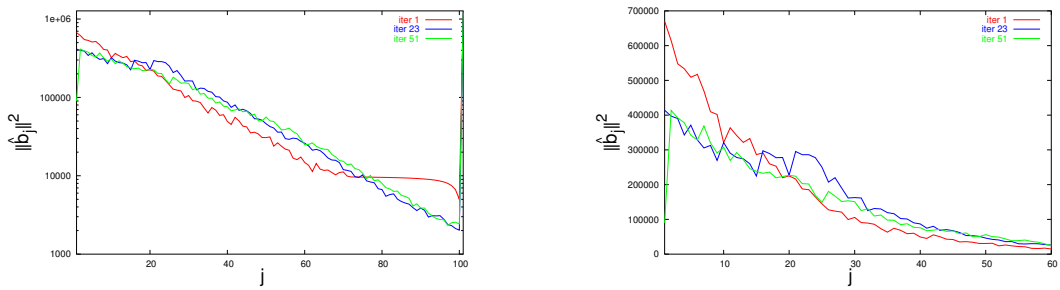
Coster, Joux, LaMacchia, Odlyzko, Schnorr, and Stern [CJL+92] advanced on this result and showed that the vector $(x_1 - 1/2, \ldots, x_n - 1/2, 0)^t$ is, with high probability, a shortest vector in the lattice generated by

$$\mathsf{B}_{\mathrm{CJLOSS}} = \begin{pmatrix} 1 & 0 & \ldots & 0 & 1/2 \\ 0 & 1 & \ldots & 0 & 1/2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 1/2 \\ Na_1 & Na_2 & \ldots & Na_n & Ns \end{pmatrix} \in \mathbb{Q}^{(n+1)\times(n+1)} \tag{6.7}$$

provided $d < 0.9408$.

Okamoto, Tanaka, and Uchiyama [OTU00] proposed a knapsack based cryptosystem to be used when the established cryptosystems can be broken in praxis by quantum computers. (Actually, their system assumes the availability of quantum computers for key generation.) Since the knapsacks in their system have density 1 and larger, they claimed their system to be secure against classical attacks.

However, it is mentioned in [CJL+92] that a minor modification of $\mathsf{B}_{\mathrm{CJLOSS}}$ allows the solution of most knapsack problems with even larger densities by an SVP oracle as long as the solution vector has low Hamming weight $k := x_1 + \cdots + x_n < n/2$. Oomura and Tanaka [OT03] give a sufficient condition that the knapsack solution corresponds, with high probability, to a shortest lattice vector. This condition can be numerically evaluated; suffice it to say that Okamoto's, Tanaka's, and Uchiyama's system is in fact vulnerable to lattice basis reduction attacks because the solution vectors of the knapsacks involved have a very low Hamming weight.



(a) Logarithmic overview over Gram-Schmidt vector lengths.

(b) Details of Gram-Schmidt vector lengths at front of basis.

Figure 6.4.: Knapsack lattice basis under ShortProjectionSR ($n = 100$, $b = 102$, $k = 10$, $d = 0.9804$, $\beta = 5$)

To solve knapsack problems with densities > 0.9408, we consider the lattice generated by

$$\mathsf{B}'_{\text{CJLOSS}} = \begin{pmatrix} n & 0 & \dots & 0 & k \\ 0 & n & \dots & 0 & k \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & n & k \\ nNa_1 & nNa_2 & \dots & nNa_n & nNs \end{pmatrix} \in \mathbb{Z}^{(n+1)\times(n+1)} . \tag{6.8}$$

The target vector $\mathbf{t} = (nx_1 - k, \dots, nx_n - k, 0)^t \in \mathsf{B}'_{\text{CJLOSS}}$ has length $\|\mathbf{t}\| = \sqrt{kn(n-k)}$ and is likely to be a shortest vector.

6.3.2. Sampling Reduction of Knapsack Lattice Bases

We describe how Sampling Reduction – here ShortProjectionSR – performs if applied to lattice bases of the form (6.8).

For our experiments, we generated knapsack instances and the corresponding lattice bases as follows: We chose the number n of weights, the maximum bitlength b of all weights, and the Hamming weight k of the solution vector $\mathbf{x}$. We then picked uniform random weights $a_1, \dots, a_n \in_R \{ 1, \dots, 2^b - 1 \}$ and a uniform random solution vector $\mathbf{x} \in_R \{ (y_1, \dots, y_n)^t \in \{ 0, 1 \}^n \mid y_1 + \dots + y_n = k \}$. From that we computed the knapsack sum $s = \sum_{j=1}^n x_j a_j$. The expected density d of the knapsack $(a_1, \dots, a_n, s)$ is $n/b < d < n/(b-1)$. In the construction of the lattice basis we always used $N = \lfloor \sqrt{n} \rfloor + 1$.

Depending on d and k, BKZ often fails to recover a shortest vector in the knapsack lattices for moderate values of β, say $\beta \leq 15$, if $n \approx 100$. We therefore tested Sampling Reduction for such knapsacks with $0.9 < d < 1.0$.

The example depicted in Fig. 6.4 on the facing page was generated with $n = 100$, $b = 102$, and $k = 10$, resulting in the density $d = 0.9804$ and the squared target vector length $\|\mathbf{t}\|^2 = 9{\times}10^4$. The red graphs show the Gram-Schmidt vector lengths after BKZ reduction of the knapsack lattice basis according to (6.8) with $(\delta, \beta) = (0.99, 5)$. They approximate a geometric sequence for the first 70 vectors only, then they stagnate. The last Gram-Schmidt vector is extraordinarily long; it is even longer than $\mathbf{b}_1$. We observed this behavior for all knapsack lattices.

It is clear, because of (2.2), that SSR as stated in Alg. 3 cannot find a vector shorter than $\mathbf{b}_1$: We have for any sample $\mathbf{v} = \hat{\mathsf{B}}(\nu_1, \ldots, \nu_n)^t$ that $\|\mathbf{v}\| \geq \nu_n \|\hat{\mathbf{b}}_n\| = \|\hat{\mathbf{b}}_n\| > \|\mathbf{b}_1\|$. On the other hand, for the same reason the target vector $\mathbf{t}$ has to belong to the lattice generated by $[\mathbf{b}_1, \ldots, \mathbf{b}_{n-1}]$: For any lattice vector $\mathbf{v}$, $\hat{\mathbf{b}}_n$ contributes an integral multiple. If $\mathbf{v} = \hat{\mathsf{B}}(\nu_1, \ldots, \nu_n)^t$ is shorter than $\hat{\mathbf{b}}$, then $\nu_n = 0$.

We therefore sampled vectors from the search space $V_{\mathbf{c},\mathsf{B}}$ with $c_n = 0$, $c_j = 1$ for $j < n - 22$, and $c_j = 2$ else. PoolSR alone did never succeed in solving the knapsack. The reduction stalled after few iterations. Interactive ShortProjectionSR, similar to the reduction described in Sect. 3.6.2, often succeeded. In this particular example, we started with reduction factor $\gamma = 0.90$ and target set $T = \{\, 1 \,\}$. We then gradually increased T up to $\{\, 1, \ldots, 16 \,\}$ in the 23rd iteration. By then the graph of the Gram-Schmidt vector lengths exhibited a bend near $j = 20$ and, for $20 < j < 60$, $\|\hat{\mathbf{b}}_j\|^2$ was consistently about 10 % larger than in the first iteration, cf. diagram 6.4(a). Since we did not find vectors any more that could be inserted at the very front of the basis, we replaced the reduction factor with $\gamma = 0.70$ and the target set with $T = \{\, j, j + 10 \,\}$, $j = 20, 30, 40$. By this we could straighten out the bend and thereby increase the chance to find a vector shorter than $\mathbf{b}_1$. Finally, after two further reductions with $\gamma = 0.90$ and $T = \{\, 1, \ldots, 5 \,\}$, ShortProjectionSR found the target vector $\mathbf{t}$. The total runtime of the reduction was 1004 seconds.

BKZ alone did not find the target vector in the same lattice unless we increased β to 30, when the reduction took 2500 seconds. We had to increase the BKZ parameter to $\beta > 25$ for most knapsacks. The runtime advantage of ShortProjectionSR we have seen here was not consistent throughout our experiments, though; in some instances, BKZ was two to three times faster than Sampling Reduction.

As in Sect. 6.2.2, however, we claim that Sampling Reduction will have an advantage with a massively distributed implementation. In all experiments, the time ShortProjectionSR spent on the intermittent BKZ reductions with small β was negligible compared to the sampling time. (In the example above it was always below our timer resolution of 1 second.) The speedup of a distributed implementation will therefore be near to proportional to the number of nodes.

Appendix A.

Sampling Results

We cannot print here the data for all of the more than 1000 experiments we ran with the Sampling Reduction algorithms presented in Chapter 3. We therefore have to restrict ourselves to a representative selection.

t_{BKZ} is the time spent on the BKZ reduction in the respective iteration, t_{sample} the time spent on sampling. All times are in seconds.

The column $u_{\min}$ refers to the the required search space size estimated by $\mathsf{CSSS}_{\mathsf{event}}$, i. e., $u_{\min} = -\max_{k \in T} \mathsf{LogSuccessProbBound}(\ell, k, u_{\max}, \|\mathbf{b}_1\|^2)$. In comparison, the value $\log_2$ #samples is the size of the search space that was actually searched.

If the algorithm is PoolSR, then m is the number vectors in the pool when the sampling loop is left. $\mathbf{p}_1$ is the shortest, $\mathbf{p}_m$ the longest pool vector.

The value e indicates at which column a sampled vector was inserted in ShortProjectionSR.

A.1. Micciancio Attack Keys

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	29	29	0	7.00	9	1258.19	$8.024\,116\times10^7$
2	32	3	0	7.00	8	1242.74	$6.851\,578\times10^7$
3	38	5	1	7.00	11	1233.79	$6.297\,051\times10^7$
4	43	4	1	9.32	14	1232.34	$6.228\,787\times10^7$
5	46	2	1	8.58	14	1230.05	$5.810\,852\times10^7$
6	48	2	0	10.58	16	1220.92	$5.387\,898\times10^7$
7	51	2	1	12.81	18	1217.72	$4.900\,536\times10^7$
8	55	3	1	14.10	20	1212.87	$4.728\,519\times10^7$
9	61	4	2	13.43	21	1200.47	$4.357\,437\times10^7$
10	80	2	17	17.14	23	1197.25	$4.067\,387\times10^7$
11	209	3	126	20.01	25	1190.77	$4.006\,410\times10^7$
12	283	2	72	19.40	26	1186.68	$3.951\,408\times10^7$
13	307	3	21	17.44	26	1189.98	$3.702\,361\times10^7$
14	527	5	215	20.81	28	1181.78	$3.572\,570\times10^7$
15	550	2	21	17.35	29	1182.07	$3.335\,083\times10^7$
16	909	2	357	21.50	32	1177.41	$2.761\,356\times10^7$

Table A.1.: SSR of Micciancio key in dimension 160
($\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	16	16	0	7.00	12	1418.58	$1.052\,824\times10^8$
2	20	4	0	10.17	13	1409.20	$9.544\,766\times10^7$
3	28	7	1	9.00	19	1405.73	$8.545\,106\times10^7$
4	57	5	24	17.73	22	1392.34	$8.213\,067\times10^7$
5	64	6	1	13.00	23	1383.66	$7.790\,800\times10^7$
6	73	4	5	15.13	24	1381.06	$7.700\,596\times10^7$
7	82	7	2	14.40	24	1366.49	$7.419\,051\times10^7$
8	120	2	36	18.14	24	1361.69	$7.294\,028\times10^7$
9	139	5	14	16.90	24	1359.38	$6.988\,381\times10^7$
10	145	3	3	14.00	25	1355.22	$6.785\,586\times10^7$
11	270	7	118	19.83	26	1349.84	$6.568\,462\times10^7$
12	356	4	82	19.42	27	1344.23	$6.488\,152\times10^7$
13	408	4	48	18.46	27	1334.85	$6.168\,118\times10^7$
14	692	3	281	21.03	29	1338.41	$6.076\,426\times10^7$
15	701	2	7	15.62	29	1331.75	$5.767\,976\times10^7$

Table A.2.: SSR of Micciancio key in dimension 170
($\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	9	9	0	10.58	14	1591.76	$1.298\,191\times10^8$
2	14	5	0	9.32	13	1583.90	$1.854\,113\times10^8$
3	23	9	0	8.00	14	1577.43	$1.821\,594\times10^8$
4	42	19	0	7.00	14	1553.25	$1.627\,421\times10^8$
5	44	2	0	10.81	15	1555.50	$1.581\,667\times10^8$
6	49	5	0	8.00	16	1556.03	$1.553\,158\times10^8$
7	53	3	1	11.46	16	1546.33	$1.507\,239\times10^8$
8	57	4	0	9.81	17	1542.63	$1.482\,750\times10^8$
9	61	4	0	7.00	17	1540.79	$1.405\,675\times10^8$
10	65	4	0	8.58	19	1532.20	$1.315\,314\times10^8$
11	75	7	3	14.35	20	1528.57	$1.295\,805\times10^8$
12	83	2	6	15.14	20	1529.47	$1.280\,163\times10^8$
13	95	2	10	15.38	21	1527.13	$1.245\,358\times10^8$
14	100	4	1	12.93	21	1519.33	$1.176\,217\times10^8$
15	108	2	6	14.70	23	1512.20	$1.052\,046\times10^8$
16	206	4	94	19.07	27	1507.82	$1.003\,334\times10^8$
17	243	5	32	17.28	29	1506.56	$9.797\,483\times10^7$
18	394	1	150	19.60	29	1503.55	$9.431\,561\times10^7$

Table A.3.: SSR of Micciancio key in dimension 180
($\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 20$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	39	39	0	7.00	10	1783.72	$3.734\,055\times10^8$
2	68	29	0	7.00	7	1780.64	$3.300\,627\times10^8$
3	82	14	0	7.00	10	1771.53	$3.337\,847\times10^8$
4	92	10	0	7.00	9	1769.00	$3.457\,063\times10^8$
5	95	3	0	7.00	11	1760.61	$2.280\,664\times10^8$
6	117	10	12	15.60	23	1740.26	$2.210\,553\times10^8$
7	127	6	4	14.37	22	1725.20	$2.158\,735\times10^8$
8	142	7	8	15.03	22	1714.61	$2.072\,194\times10^8$
9	177	5	30	17.06	22	1711.03	$1.929\,402\times10^8$
10	218	3	38	17.21	25	1709.73	$1.701\,482\times10^8$
11	1176	3	958	22.00	29	1709.73	$1.701\,482\times10^8$

Table A.4.: SSR of Micciancio key in dimension 190
($\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	25	25	0	7.00	10	1980.28	$5.767\,064\times10^8$
2	48	22	1	9.32	11	1944.24	$4.953\,965\times10^8$
3	75	27	0	8.00	13	1922.24	$4.701\,614\times10^8$
4	84	9	0	8.00	12	1918.64	$4.279\,279\times10^8$
5	91	6	1	10.46	16	1914.87	$4.167\,087\times10^8$
6	107	15	1	9.32	16	1906.86	$3.892\,371\times10^8$
7	119	9	3	12.75	17	1901.38	$3.089\,206\times10^8$
8	246	5	122	18.81	25	1900.32	$3.054\,658\times10^8$
9	257	5	6	14.15	24	1884.88	$3.021\,978\times10^8$
10	273	10	6	14.51	25	1882.76	$2.921\,377\times10^8$
11	626	9	344	20.26	25	1871.16	$2.838\,058\times10^8$
12	890	8	256	19.87	26	1862.57	$2.806\,805\times10^8$
13	971	11	70	18.00	26	1856.74	$2.493\,063\times10^8$
14	2134	11	1163	22.00	30	1856.74	$2.493\,063\times10^8$

Table A.5.: SSR of Micciancio key in dimension 200
($\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$	m	$\lVert \mathbf{p}_1 \rVert^2$	$\lVert \mathbf{p}_m \rVert^2$
1	16	16	0	10.58	14	1587.02	$1.974\,042\times10^8$	5	$1.974\,042\times10^8$	$2.136\,649\times10^8$
2	34	17	1	10.91	13	1569.66	$1.511\,494\times10^8$	5	$1.926\,642\times10^8$	$2.016\,907\times10^8$
3	45	11	0	9.00	11	1554.30	$1.791\,383\times10^8$	5	$1.874\,813\times10^8$	$2.022\,695\times10^8$
4	52	7	0	7.00	11	1539.20	$1.715\,853\times10^8$	1	$1.715\,853\times10^8$	$1.715\,853\times10^8$
5	65	13	0	7.00	12	1537.15	$1.405\,920\times10^8$	2	$1.678\,752\times10^8$	$1.759\,390\times10^8$
6	73	8	0	8.58	12	1533.69	$1.567\,843\times10^8$	5	$1.567\,843\times10^8$	$1.908\,604\times10^8$
7	77	3	1	8.00	14	1534.35	$1.406\,282\times10^8$	1	$1.406\,282\times10^8$	$1.406\,282\times10^8$
8	82	4	1	11.17	18	1535.16	$1.265\,281\times10^8$	2	$1.265\,281\times10^8$	$1.539\,811\times10^8$
9	100	8	10	15.68	22	1528.61	$1.231\,828\times10^8$	3	$1.231\,828\times10^8$	$1.353\,657\times10^8$
10	107	6	1	8.58	22	1515.26	$1.106\,481\times10^8$	2	$1.106\,481\times10^8$	$1.233\,101\times10^8$
11	303	8	188	19.89	25	1498.32	$9.255\,114\times10^7$	5	$9.255\,114\times10^7$	$1.199\,898\times10^8$
12	1086	8	783	22.00	31	1498.32	$9.255\,114\times10^7$	1	$9.550\,509\times10^7$	$9.550\,509\times10^7$

Table A.6.: PoolSR of Micciancio key in dimension 180
($\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 20$, poolsize 5, $\alpha = 25$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2 \#\text{samples}$	$u_{\min}$	$\log_2 \text{odef}(\text{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	m	$\lVert \mathbf{p}_1 \rVert^2$	$\lVert \mathbf{p}_m \rVert^2$
1	53	52	1	10.58	14	1591.89	$1.974\,042\times10^8$	60	$1.974\,042\times10^8$	$2.652\,139\times10^8$
2	120	66	1	9.00	14	1579.33	$1.823\,686\times10^8$	60	$1.823\,686\times10^8$	$2.837\,015\times10^8$
3	157	36	1	7.00	14	1563.87	$1.703\,851\times10^8$	59	$1.703\,851\times10^8$	$4.179\,686\times10^8$
4	222	64	1	9.58	15	1538.95	$1.534\,517\times10^8$	60	$1.534\,517\times10^8$	$2.419\,111\times10^8$
5	261	38	1	11.00	16	1542.68	$1.506\,308\times10^8$	60	$1.506\,308\times10^8$	$2.017\,231\times10^8$
6	309	47	1	8.00	19	1537.32	$1.354\,623\times10^8$	60	$1.354\,623\times10^8$	$2.685\,572\times10^8$
7	385	59	17	16.30	21	1523.58	$1.257\,034\times10^8$	60	$1.257\,034\times10^8$	$1.622\,141\times10^8$
8	542	81	76	18.47	23	1504.93	$1.146\,994\times10^8$	60	$1.146\,994\times10^8$	$1.409\,539\times10^8$
9	603	38	23	16.90	23	1487.06	$1.073\,186\times10^8$	60	$1.073\,186\times10^8$	$1.398\,544\times10^8$
10	765	48	114	19.17	26	1489.69	$1.047\,262\times10^8$	60	$1.047\,262\times10^8$	$1.267\,841\times10^8$
11	1390	71	554	21.64	29	1474.79	$9.627\,016\times10^7$	60	$9.627\,016\times10^7$	$1.200\,592\times10^8$
12	1534	64	80	18.72	29	1465.13	$9.280\,497\times10^7$	60	$9.280\,497\times10^7$	$1.247\,564\times10^8$
13	1894	60	300	20.76	29	1457.67	$9.159\,888\times10^7$	60	$9.159\,888\times10^7$	$1.098\,694\times10^8$
14	2499	58	547	21.39	30	1453.71	$8.834\,502\times10^7$	60	$8.834\,502\times10^7$	$1.099\,308\times10^8$
15	3247	58	748	22.00	32	1453.71	$8.834\,502\times10^7$	60	$9.075\,550\times10^7$	$1.070\,317\times10^8$

Table A.7.: PoolSR of Micciancio key in dimension 180
($\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 20$, poolsize 60, $\alpha = 1$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert\mathbf{b}_j\rVert^2$	m	$\lVert\mathbf{p}_1\rVert^2$	$\lVert\mathbf{p}_m\rVert^2$
1	18	18	0	10.58	14	1587.90	$1.974\,042\times10^8$	4	$1.974\,042\times10^8$	$2.231\,460\times10^8$
2	35	17	0	10.17	12	1577.01	$1.826\,020\times10^8$	10	$1.826\,020\times10^8$	$2.472\,612\times10^8$
3	54	18	1	7.00	14	1559.52	$1.770\,224\times10^8$	2	$1.770\,224\times10^8$	$1.819\,422\times10^8$
4	63	9	0	8.00	13	1536.02	$1.635\,720\times10^8$	4	$1.635\,720\times10^8$	$2.112\,397\times10^8$
5	72	9	0	9.00	13	1529.68	$1.511\,125\times10^8$	1	$1.511\,125\times10^8$	$1.511\,125\times10^8$
6	82	10	0	9.81	15	1521.04	$1.394\,176\times10^8$	2	$1.394\,176\times10^8$	$1.555\,224\times10^8$
7	92	8	2	13.17	17	1514.73	$1.379\,422\times10^8$	7	$1.379\,422\times10^8$	$1.564\,616\times10^8$
8	95	3	0	8.00	17	1516.46	$1.222\,541\times10^8$	2	$1.222\,541\times10^8$	$1.395\,758\times10^8$
9	100	5	0	9.81	21	1506.99	$1.195\,460\times10^8$	2	$1.195\,460\times10^8$	$1.315\,225\times10^8$
10	123	15	8	15.47	22	1495.06	$1.174\,098\times10^8$	5	$1.174\,098\times10^8$	$1.258\,241\times10^8$
11	139	5	11	15.92	22	1499.54	$1.112\,644\times10^8$	5	$1.112\,644\times10^8$	$1.393\,693\times10^8$
12	188	11	38	17.60	24	1492.58	$9.982\,749\times10^7$	5	$9.982\,749\times10^7$	$1.349\,456\times10^8$
13	243	6	49	17.98	27	1488.70	$8.621\,029\times10^7$	3	$8.621\,029\times10^7$	$1.027\,476\times10^8$
14	286	6	37	17.65	33	1480.97	$8.267\,564\times10^7$	1	$8.267\,564\times10^7$	$8.267\,564\times10^7$

Table A.8.: PoolSR of Micciancio key in dimension 180 ($\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 20$, poolsize 60, $\alpha = 25$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	m	$\lVert \mathbf{p}_1 \rVert^2$	$\lVert \mathbf{p}_m \rVert^2$
1	53	53	0	7.00	10	1949.57	$5.019\,617\times10^8$	3	$5.969\,302\times10^8$	$6.466\,930\times10^8$
2	73	20	0	8.58	10	1930.92	$4.510\,970\times10^8$	2	$4.827\,592\times10^8$	$5.623\,965\times10^8$
3	84	11	0	8.58	13	1916.14	$4.355\,080\times10^8$	1	$4.620\,905\times10^8$	$4.620\,905\times10^8$
4	99	14	1	9.00	13	1908.38	$3.874\,272\times10^8$	2	$4.561\,172\times10^8$	$4.650\,428\times10^8$
5	107	8	0	8.58	18	1897.45	$3.487\,939\times10^8$	1	$3.487\,939\times10^8$	$3.487\,939\times10^8$
6	118	10	1	9.00	20	1894.81	$3.233\,435\times10^8$	1	$3.233\,435\times10^8$	$3.233\,435\times10^8$
7	157	14	25	16.51	23	1887.22	$3.002\,872\times10^8$	2	$3.002\,872\times10^8$	$3.692\,301\times10^8$
8	271	8	106	18.61	25	1870.40	$2.910\,530\times10^8$	5	$2.910\,530\times10^8$	$3.081\,754\times10^8$
9	399	14	114	18.65	26	1860.87	$2.872\,567\times10^8$	4	$2.872\,567\times10^8$	$3.447\,302\times10^8$
10	523	9	115	18.75	26	1860.40	$2.826\,705\times10^8$	5	$2.826\,705\times10^8$	$3.529\,975\times10^8$
11	565	18	24	16.45	27	1843.32	$2.775\,427\times10^8$	3	$2.775\,427\times10^8$	$3.364\,603\times10^8$
12	631	12	54	17.51	27	1837.01	$2.653\,174\times10^8$	1	$2.653\,174\times10^8$	$2.653\,174\times10^8$
13	1073	17	425	20.57	28	1828.59	$2.580\,287\times10^8$	2	$2.580\,287\times10^8$	$2.678\,458\times10^8$
14	1227	7	147	19.01	27	1820.40	$2.446\,416\times10^8$	4	$2.446\,416\times10^8$	$3.007\,448\times10^8$
15	1419	7	185	19.44	29	1809.58	$2.334\,918\times10^8$	3	$2.334\,918\times10^8$	$2.554\,872\times10^8$
16	1606	12	175	19.27	30	1804.28	$2.120\,052\times10^8$	2	$2.120\,052\times10^8$	$2.530\,603\times10^8$
17	2377	12	759	21.33	33	1804.01	$1.997\,645\times10^8$	1	$1.997\,645\times10^8$	$1.997\,645\times10^8$
18	3536	12	1159	22.00	36	1804.01	$1.997\,645\times10^8$	1	$2.143\,745\times10^8$	$2.143\,745\times10^8$

Table A.9.: PoolSR of Micciancio key in dimension 200 ($\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 22$, poolsize 60, $\alpha = 25$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
1	7	7	0	11.46	15	1595.04	$1.751\,650\times10^8$	1
2	30	22	1	9.32	18	1575.07	$1.575\,861\times10^8$	1
3	48	15	3	14.23	19	1556.51	$1.382\,076\times10^8$	1
4	60	6	6	15.01	21	1541.42	$1.063\,688\times10^8$	1
6	66	5	1	7.00	6	1538.83	$1.063\,688\times10^8$	21
7	69	2	1	7.00	8	1536.51	$1.063\,688\times10^8$	21
8	71	2	0	7.00	8	1535.49	$1.063\,688\times10^8$	21
9	76	5	0	7.00	7	1530.11	$1.063\,688\times10^8$	21
10	78	2	0	7.00	8	1529.53	$1.063\,688\times10^8$	24
11	86	7	1	7.00	10	1520.63	$1.063\,688\times10^8$	21
12	92	5	1	7.00	9	1523.77	$1.063\,688\times10^8$	24
13	95	3	0	7.00	10	1519.03	$1.063\,688\times10^8$	21
14	99	4	0	7.00	8	1518.52	$1.063\,688\times10^8$	23
15	103	4	0	7.00	9	1515.65	$1.063\,688\times10^8$	21
16	108	5	0	7.00	7	1514.26	$1.063\,688\times10^8$	21
17	114	5	1	7.00	10	1514.31	$1.063\,688\times10^8$	21
18	116	2	0	7.00	10	1506.42	$1.063\,688\times10^8$	23
19	120	4	0	7.00	11	1508.16	$1.063\,688\times10^8$	21
20	126	5	1	8.00	11	1509.53	$1.063\,688\times10^8$	21
21	129	3	0	7.00	9	1502.94	$1.063\,688\times10^8$	22
22	134	5	0	8.00	10	1505.63	$1.063\,688\times10^8$	21
23	139	5	0	7.00	11	1503.95	$1.063\,688\times10^8$	22
24	140	1	0	7.00	11	1507.95	$1.063\,688\times10^8$	21
25	142	1	1	8.00	12	1499.17	$1.063\,688\times10^8$	21
26	145	2	1	7.00	12	1497.89	$1.063\,688\times10^8$	25
27	151	5	1	7.00	12	1500.06	$1.063\,688\times10^8$	21
28	155	3	1	7.00	10	1494.81	$1.063\,688\times10^8$	21
29	159	4	0	7.00	10	1497.51	$1.063\,688\times10^8$	22
30	161	2	0	7.00	9	1501.10	$1.063\,688\times10^8$	22
31	164	3	0	7.00	8	1501.54	$1.063\,688\times10^8$	24
32	167	2	1	7.00	7	1491.51	$1.063\,688\times10^8$	21
33	170	2	1	7.00	12	1487.65	$1.063\,688\times10^8$	21
34	173	2	1	7.00	11	1492.92	$1.063\,688\times10^8$	23
35	176	3	0	9.00	12	1488.84	$1.063\,688\times10^8$	23
36	179	3	0	7.00	12	1486.83	$1.063\,688\times10^8$	25
37	182	3	0	8.00	12	1491.18	$1.063\,688\times10^8$	23
38	185	2	1	7.00	11	1489.21	$1.063\,688\times10^8$	25
39	187	2	0	8.00	12	1486.85	$1.063\,688\times10^8$	21
40	191	4	0	8.00	12	1484.88	$1.063\,688\times10^8$	22
41	194	3	0	7.00	13	1481.41	$1.063\,688\times10^8$	24
42	196	2	0	8.00	11	1483.49	$1.063\,688\times10^8$	25
43	199	3	0	8.00	17	1478.31	$1.063\,688\times10^8$	25
44	203	3	1	8.00	17	1477.33	$1.063\,688\times10^8$	21

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
45	206	2	1	9.32	14	1481.25	$1.063\,688\times10^8$	21
46	209	3	0	9.81	15	1477.64	$1.063\,688\times10^8$	21
47	211	1	1	7.00	14	1477.19	$1.063\,688\times10^8$	24
48	214	3	0	7.00	15	1474.66	$1.063\,688\times10^8$	21
49	217	3	0	7.00	13	1471.61	$1.063\,688\times10^8$	22
50	219	2	0	7.00	12	1475.65	$1.063\,688\times10^8$	24
51	221	1	1	8.58	11	1477.20	$1.063\,688\times10^8$	25
52	224	2	1	10.46	15	1477.56	$1.063\,688\times10^8$	23
53	226	2	0	10.00	15	1479.34	$1.063\,688\times10^8$	21
54	229	2	1	9.00	16	1480.71	$1.063\,688\times10^8$	22
55	232	2	1	10.00	15	1471.95	$1.063\,688\times10^8$	23
56	234	2	0	8.58	13	1478.25	$1.063\,688\times10^8$	21
57	237	2	1	9.32	12	1474.84	$1.063\,688\times10^8$	25
58	239	2	0	11.09	15	1476.87	$1.063\,688\times10^8$	25
59	242	3	0	8.00	17	1470.74	$1.063\,688\times10^8$	22
60	243	0	1	9.58	17	1473.48	$1.063\,688\times10^8$	21
61	245	2	0	11.25	15	1474.68	$1.063\,688\times10^8$	24
62	247	1	1	10.00	18	1471.27	$1.063\,688\times10^8$	25
63	251	1	3	13.48	18	1473.12	$1.063\,688\times10^8$	23
64	253	2	0	9.58	19	1476.20	$1.063\,688\times10^8$	21
65	256	1	2	12.09	18	1469.01	$1.063\,688\times10^8$	22
66	260	2	2	12.93	18	1468.65	$1.063\,688\times10^8$	24
67	263	1	2	13.07	18	1469.15	$1.063\,688\times10^8$	25
68	266	2	1	9.32	19	1468.86	$1.063\,688\times10^8$	21
69	269	2	1	9.58	18	1464.79	$1.063\,688\times10^8$	25
70	272	2	1	12.17	19	1462.68	$1.063\,688\times10^8$	21
71	274	1	1	12.13	19	1467.06	$1.063\,688\times10^8$	25
72	277	2	1	10.00	19	1466.07	$1.063\,688\times10^8$	25
73	280	2	1	12.36	19	1462.32	$1.063\,688\times10^8$	23
74	281	1	0	7.00	19	1463.14	$1.063\,688\times10^8$	22
76	284	2	0	7.00	10	1460.20	$1.063\,688\times10^8$	43
77	286	1	1	7.00	10	1466.35	$1.063\,688\times10^8$	44
78	289	2	1	8.58	13	1463.70	$1.063\,688\times10^8$	42
79	291	1	1	9.58	14	1463.21	$1.063\,688\times10^8$	41
80	293	2	0	8.00	13	1464.51	$1.063\,688\times10^8$	41
81	294	1	0	8.00	14	1464.39	$1.063\,688\times10^8$	44
82	298	3	1	10.58	14	1466.47	$1.063\,688\times10^8$	45
83	301	2	1	10.70	17	1460.80	$1.063\,688\times10^8$	44
84	304	2	1	11.17	16	1463.15	$1.063\,688\times10^8$	41
85	307	3	0	10.58	17	1463.27	$1.063\,688\times10^8$	44
86	311	2	2	12.83	16	1462.04	$1.063\,688\times10^8$	45
87	313	2	0	9.58	20	1463.24	$1.063\,688\times10^8$	41
88	316	2	1	11.17	17	1456.24	$1.063\,688\times10^8$	41

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
89	318	1	1	7.00	16	1460.44	$1.063\,688\times10^8$	42
90	320	1	1	10.46	16	1462.99	$1.063\,688\times10^8$	43
91	323	2	1	11.09	15	1457.83	$1.063\,688\times10^8$	45
92	326	2	1	12.39	19	1463.32	$1.063\,688\times10^8$	44
93	328	1	1	10.58	20	1460.17	$1.063\,688\times10^8$	42
94	332	3	1	11.70	19	1462.44	$1.063\,688\times10^8$	41
95	334	2	0	8.00	18	1463.91	$1.063\,688\times10^8$	44
96	336	1	1	11.17	17	1462.55	$1.063\,688\times10^8$	41
97	339	2	1	10.58	17	1463.49	$1.063\,688\times10^8$	43
98	342	2	1	12.61	18	1461.72	$1.063\,688\times10^8$	45
99	344	1	1	10.91	19	1461.50	$1.063\,688\times10^8$	42
100	347	2	1	11.32	18	1464.84	$1.063\,688\times10^8$	44
101	349	2	0	10.00	15	1460.92	$1.063\,688\times10^8$	44
102	351	1	1	8.00	13	1458.04	$1.063\,277\times10^8$	45
103	354	2	1	12.55	20	1463.76	$1.063\,688\times10^8$	42
104	356	2	0	9.32	18	1458.89	$1.063\,688\times10^8$	43
105	360	2	2	13.17	20	1461.21	$1.063\,688\times10^8$	45
106	363	1	2	12.39	20	1459.78	$1.063\,688\times10^8$	45
108	365	2	0	7.00	11	1461.22	$1.063\,688\times10^8$	62
109	367	2	0	7.00	11	1461.10	$1.063\,688\times10^8$	61
110	368	1	0	7.00	10	1454.86	$1.063\,688\times10^8$	62
111	369	1	0	7.00	12	1459.17	$1.063\,688\times10^8$	63
112	372	2	1	9.81	14	1460.09	$1.063\,688\times10^8$	65
113	373	1	0	8.58	15	1458.49	$1.063\,688\times10^8$	62
114	375	2	0	7.00	13	1456.71	$1.063\,688\times10^8$	63
115	376	0	1	8.58	16	1459.52	$1.063\,688\times10^8$	65
116	378	1	1	9.00	16	1464.69	$1.063\,688\times10^8$	61
117	379	1	0	7.00	13	1459.26	$1.063\,688\times10^8$	61
118	383	3	1	7.00	12	1457.38	$1.063\,688\times10^8$	61
119	384	1	0	8.58	13	1458.74	$1.063\,688\times10^8$	63
120	385	1	0	8.58	15	1462.37	$1.063\,688\times10^8$	65
121	388	2	1	9.81	17	1459.27	$1.063\,688\times10^8$	62
122	390	1	1	12.21	17	1459.58	$1.063\,688\times10^8$	63
123	391	1	0	10.17	16	1457.94	$1.063\,688\times10^8$	64
124	393	2	0	10.00	15	1452.87	$1.063\,688\times10^8$	65
125	395	1	1	12.46	19	1453.57	$1.063\,688\times10^8$	63
126	397	2	0	11.75	19	1455.92	$1.063\,688\times10^8$	63
127	400	3	0	8.58	18	1454.20	$1.063\,688\times10^8$	62
128	402	1	1	11.17	17	1455.88	$1.063\,688\times10^8$	63
129	404	2	0	10.58	17	1460.64	$1.063\,688\times10^8$	64
130	407	2	1	11.91	17	1456.36	$1.063\,688\times10^8$	65
131	411	1	3	13.39	20	1461.69	$1.063\,688\times10^8$	61
132	413	2	0	11.58	19	1456.93	$1.063\,688\times10^8$	65

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
133	415	1	1	12.73	18	1453.63	$1.063\,688\times10^8$	61
134	418	1	2	12.67	18	1456.29	$1.063\,688\times10^8$	62
135	419	1	0	7.00	17	1456.42	$1.063\,688\times10^8$	63
136	422	2	1	11.75	18	1457.81	$1.063\,688\times10^8$	61
137	425	2	1	9.58	17	1456.27	$1.063\,688\times10^8$	65
138	426	0	1	10.91	19	1456.31	$1.063\,688\times10^8$	65
140	428	2	0	11.81	19	1456.81	$9.221\,797\times10^7$	1
141	460	2	30	17.22	24	1457.18	$8.543\,042\times10^7$	1
143	462	1	1	9.32	19	1451.36	$8.543\,042\times10^7$	23
144	469	2	5	14.40	19	1458.42	$8.543\,042\times10^7$	21
145	472	2	1	8.00	17	1453.20	$8.543\,042\times10^7$	21
146	475	2	1	11.17	17	1455.30	$8.543\,042\times10^7$	22
147	477	2	0	9.00	16	1455.77	$8.543\,042\times10^7$	23
148	480	2	1	9.81	16	1457.05	$8.543\,042\times10^7$	24
149	482	2	0	9.58	17	1459.16	$8.543\,042\times10^7$	25
150	483	1	0	8.00	19	1462.00	$8.543\,042\times10^7$	23
151	485	1	1	9.00	17	1454.99	$8.543\,042\times10^7$	25
152	487	1	1	10.00	20	1457.08	$8.543\,042\times10^7$	24
153	491	1	3	13.70	19	1455.89	$8.543\,042\times10^7$	25
155	494	1	1	10.46	16	1456.74	$8.543\,042\times10^7$	41
156	496	1	1	7.00	14	1457.50	$8.543\,042\times10^7$	41
157	497	0	1	11.86	18	1452.28	$8.543\,042\times10^7$	41
158	500	2	1	8.58	17	1460.92	$8.543\,042\times10^7$	42
159	504	2	2	13.13	19	1457.67	$8.543\,042\times10^7$	42
160	507	3	0	11.00	19	1460.43	$8.543\,042\times10^7$	43
162	510	2	0	9.00	15	1453.34	$8.543\,042\times10^7$	65
163	512	1	1	9.81	16	1454.69	$8.543\,042\times10^7$	61
164	513	1	0	9.81	15	1459.09	$8.543\,042\times10^7$	65
165	515	1	1	10.32	16	1457.88	$8.543\,042\times10^7$	62
166	516	1	0	10.17	15	1457.28	$8.543\,042\times10^7$	63
167	518	1	1	10.32	16	1454.11	$8.543\,042\times10^7$	64
168	519	1	0	11.25	16	1456.78	$8.543\,042\times10^7$	65
169	521	1	1	11.91	20	1456.40	$8.543\,042\times10^7$	63
170	522	0	1	11.00	19	1454.95	$8.543\,042\times10^7$	64
171	525	1	2	13.54	19	1458.41	$8.543\,042\times10^7$	65

Table A.10.: Interactive ShortProjectionSR of Micciancio key in dimension 180 ($\delta = 0.99$, $\beta = 5$, $\gamma = 0.90$ to 0.99, $u_{\max} = 20$ to 26)

A.2. Micciancio Embedding Attacks

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	2	2	0	7.00	2	1585.44	$2.260\,091\times10^8$
2	2	0	0	7.00	13	1545.12	$1.900\,500\times10^4$

Table A.11.: SSR of embedded Micciancio ciphertext in dimension 160
($\lVert\mathbf{m}\rVert = 1.0\rho$, $\delta = 0.99$, $\beta = 2$, $\gamma = 0.99$, $u_{\max} = 20$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	3	3	0	7.00	2	1747.17	$3.522\,115\times10^8$
2	4	1	0	7.00	8	1700.64	$1.103\,100\times10^4$

Table A.12.: SSR of embedded Micciancio ciphertext in dimension 170
($\lVert\mathbf{m}\rVert = 0.8\rho$, $\delta = 0.99$, $\beta = 2$, $\gamma = 0.99$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	8	8	0	7.00	2	2060.26	$1.066\,702\times10^9$
2	11	3	0	7.00	4	1992.81	$8.020\,376\times10^8$
3	13	1	1	7.00	7	1968.42	$7.934\,961\times10^8$
4	13	0	0	7.00	7	1966.92	$7.673\,239\times10^8$
5	14	1	0	8.00	11	1956.13	$6.509\,493\times10^8$
6	16	1	1	8.00	14	1909.59	$5.012\,561\times10^8$
7	17	1	0	8.00	12	1883.93	$5.478\,531\times10^8$
8	17	0	0	8.00	13	1879.46	$5.357\,503\times10^8$
9	17	0	0	8.58	13	1880.48	$4.813\,114\times10^8$
10	19	1	1	11.95	16	1859.01	$3.870\,022\times10^8$
11	24	0	5	14.71	20	1842.16	$3.271\,083\times10^8$
12	98	0	74	18.53	24	1836.52	$3.151\,993\times10^8$
13	159	2	59	18.24	25	1836.58	$2.996\,763\times10^8$
14	200	0	41	17.84	26	1824.11	$2.962\,720\times10^8$
15	307	0	107	19.14	26	1811.09	$2.898\,377\times10^8$
16	354	1	46	17.94	26	1813.16	$2.804\,977\times10^8$
17	956	0	602	21.64	27	1808.07	$2.572\,906\times10^8$
18	1628	0	672	21.81	30	1782.53	$2.410\,850\times10^8$

Table A.13.: SSR of embedded Micciancio ciphertext in dimension 180
($\lVert\mathbf{m}\rVert = 0.9\rho$, $\delta = 0.99$, $\beta = 2$, $\gamma = 0.99$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	19	19	0	7.00	4	2195.12	$1.521\,452\times10^9$
2	21	2	0	7.00	2	2177.22	$1.170\,514\times10^9$
3	22	1	0	9.32	10	2157.71	$9.838\,076\times10^8$
4	24	1	1	9.81	17	2160.96	$9.721\,575\times10^8$
5	25	1	0	7.00	17	2142.37	$1.415\,300\times10^4$

Table A.14.: SSR of embedded Micciancio ciphertext in dimension 190
($\lVert\mathbf{m}\rVert = 0.8\rho$, $\delta = 0.99$, $\beta = 2$, $\gamma = 0.99$, $u_{\max} = 20$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	5	5	0	7.00	8	2413.48	$2.160\,723\times10^9$
2	7	2	0	9.00	10	2376.01	$2.086\,536\times10^9$
3	8	1	0	8.00	10	2357.76	$1.674\,051\times10^9$
4	10	1	1	9.58	15	2351.36	$1.649\,850\times10^9$
5	12	2	0	9.81	15	2327.83	$1.686\,400\times10^4$
6	15	1	2	12.78	15	2329.52	$1.509\,569\times10^9$
7	18	1	2	11.95	15	2322.57	$1.408\,774\times10^9$
8	20	1	1	10.17	17	2308.05	$1.325\,006\times10^9$
9	22	1	1	10.00	17	2297.07	$1.306\,589\times10^9$
10	25	1	2	12.46	17	2301.46	$1.224\,039\times10^9$
11	40	2	13	15.52	19	2291.71	$1.196\,393\times10^9$
12	50	2	8	14.81	19	2255.54	$1.182\,914\times10^9$
13	52	1	1	10.00	18	2242.85	$1.686\,400\times10^4$
14	57	3	2	12.88	18	2218.24	$1.686\,400\times10^4$

Table A.15.: SSR of embedded Micciancio ciphertext in dimension 200
($\lVert\mathbf{m}\rVert = 0.9\rho$, $\delta = 0.99$, $\beta = 2$, $\gamma = 0.99$, $u_{\max} = 20$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert\mathbf{b}_j\rVert^2$
1	141	141	0	7.00	9	3323.45	$7.962\,240\times10^9$
2	226	85	0	7.00	9	3313.47	$8.528\,386\times10^9$
3	241	15	0	8.00	10	3312.48	$7.108\,489\times10^9$
4	267	26	0	9.81	16	3277.70	$6.889\,072\times10^9$
5	284	16	1	10.17	16	3261.70	$6.485\,753\times10^9$
6	299	13	2	11.09	16	3257.15	$6.045\,238\times10^9$
7	326	15	12	14.34	19	3246.07	$5.653\,133\times10^9$
8	356	24	6	13.43	21	3230.59	$5.279\,683\times10^9$
9	386	30	0	7.00	22	3224.99	$5.199\,523\times10^9$
10	521	28	107	17.56	22	3210.48	$5.047\,799\times10^9$
11	576	31	24	15.35	22	3204.75	$4.988\,587\times10^9$
12	627	35	16	14.75	21	3196.94	$4.667\,730\times10^9$
13	674	22	25	15.41	23	3182.53	$4.564\,027\times10^9$
14	849	25	150	18.05	23	3179.73	$4.106\,910\times10^9$
15	1051	8	194	18.41	27	3182.81	$4.016\,704\times10^9$
16	1430	15	364	19.36	28	3174.64	$3.880\,310\times10^9$
17	3539	15	2094	21.83	29	3164.84	$3.787\,200\times10^9$
18	5838	15	2299	22.00	30	3164.84	$3.787\,200\times10^9$

Table A.16.: SSR of embedded Micciancio ciphertext in dimension 260
($\lVert\mathbf{m}\rVert = 1.0\rho$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert\mathbf{b}_j\rVert^2$
1	114	113	1	9.81	12	3821.48	$1.788\,397\times10^{10}$
2	240	126	0	7.00	10	3808.87	$1.768\,002\times10^{10}$
3	290	49	1	9.58	14	3798.17	$1.693\,838\times10^{10}$
4	314	23	1	8.58	15	3781.54	$1.443\,316\times10^{10}$
5	382	37	31	15.52	20	3764.57	$1.270\,081\times10^{10}$
6	480	29	69	16.68	23	3766.25	$1.119\,762\times10^{10}$
7	981	24	477	19.47	27	3743.74	$1.037\,657\times10^{10}$
8	3771	41	2749	21.99	29	3725.06	$1.017\,391\times10^{10}$
9	5072	26	1275	20.91	29	3712.81	$1.006\,892\times10^{10}$
10	5352	15	265	18.62	28	3712.17	$9.860\,259\times10^9$
11	6489	31	1106	20.70	29	3697.38	$8.470\,504\times10^9$

Table A.17.: SSR of embedded Micciancio ciphertext in dimension 280
($\lVert\mathbf{m}\rVert = 1.0\rho$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert\mathbf{b}_j\rVert^2$
1	100	99	1	7.00	9	3855.33	$1.703\,600\times10^4$

Table A.18.: SSR of embedded Micciancio ciphertext in dimension 280
($\lVert\mathbf{m}\rVert = 0.6\rho$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	127	126	1	7.00	9	4410.92	$4.208\,929\times10^{10}$
2	219	91	1	9.81	13	4386.83	$4.775\,310\times10^{10}$
3	244	25	0	8.58	13	4350.60	$3.615\,435\times10^{10}$
4	315	70	1	7.00	14	4340.49	$4.045\,268\times10^{10}$
5	384	68	1	9.00	13	4314.93	$1.196\,300\times10^{4}$

Table A.19.: SSR of embedded Micciancio ciphertext in dimension 300
($\lVert\mathbf{m}\rVert = 0.5\rho$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$
1	307	306	1	7.00	9	4374.40	$4.625\,981\times10^{10}$
2	382	74	1	7.00	13	4362.17	$4.310\,512\times10^{10}$
3	453	69	2	11.09	14	4342.12	$3.828\,078\times10^{10}$
4	494	34	7	13.11	17	4334.06	$3.719\,217\times10^{10}$
5	522	25	3	11.58	17	4339.92	$3.085\,304\times10^{10}$
6	649	38	89	16.88	23	4300.39	$3.052\,493\times10^{10}$
7	762	36	77	16.68	22	4301.31	$2.845\,876\times10^{10}$
8	974	28	184	17.92	25	4291.62	$2.755\,454\times10^{10}$
9	1107	25	108	17.15	26	4262.22	$2.685\,506\times10^{10}$
10	1493	32	354	18.87	25	4258.41	$2.571\,275\times10^{10}$
11	2575	23	1059	20.46	27	4247.94	$2.529\,258\times10^{10}$
12	3292	32	685	19.82	26	4249.79	$2.167\,593\times10^{10}$

Table A.20.: SSR of embedded Micciancio ciphertext in dimension 300
($\lVert\mathbf{m}\rVert = 1.0\rho$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.99$, $u_{\max} = 22$)

A.3. NTRU Lattices

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert\mathbf{b}_j\rVert^2$	e
1	13	13	0	8.00	10	1874.18	$2.592\,100\times10^{10}$	4
2	21	8	0	7.00	12	1853.67	$2.262\,000\times10^{9}$	1
3	100	79	0	7.00	51	972.67	$5.460\,000\times10^{8}$	1

Table A.21.: Interactive ShortProjectionSR of NTRU lattice
($N = 97$, $a = 0.602$, $c = 1.764$, $\delta = 0.99$, $\beta = 4$, $\gamma = 0.90$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert\mathbf{b}_j\rVert^2$	e
1	12	11	1	8.00	10	1999.91	$2.755\,600\times10^{10}$	8
2	17	4	1	8.58	13	1983.15	$2.755\,600\times10^{10}$	8
3	30	10	3	13.17	19	1972.66	$2.755\,600\times10^{10}$	8
4	300	7	263	19.61	27	1956.81	$1.780\,000\times10^{9}$	1
5	310	9	1	10.58	15	1932.35	$1.780\,000\times10^{9}$	10
6	314	3	1	7.00	17	1934.32	$1.780\,000\times10^{9}$	10
7	336	5	17	15.65	20	1923.90	$1.780\,000\times10^{9}$	10
8	428	16	76	17.95	22	1914.81	$1.780\,000\times10^{9}$	10
9	431	1	2	11.52	24	1909.06	$1.780\,000\times10^{9}$	2
10	436	3	2	11.09	23	1890.03	$1.780\,000\times10^{9}$	3
11	442	5	1	10.00	22	1878.69	$1.780\,000\times10^{9}$	4
12	445	2	1	7.00	20	1871.79	$1.780\,000\times10^{9}$	4
13	450	4	1	8.00	19	1844.46	$1.780\,000\times10^{9}$	6
14	527	76	1	7.00	17	1035.10	$5.600\,000\times10^{8}$	6

Table A.22.: Interactive ShortProjectionSR of NTRU lattice
($N = 100$, $a = 0.602$, $c = 1.780$, $\delta = 0.99$, $\beta = 3$, $\gamma = 0.90$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert\mathbf{b}_j\rVert^2$	e
1	31	31	0	7.00	5	2197.49	$2.924\,100\times10^{10}$	6
2	40	9	0	7.00	11	2138.31	$2.924\,100\times10^{10}$	6
3	51	9	2	12.17	17	2134.45	$2.924\,100\times10^{10}$	6
4	58	4	3	13.04	20	2125.76	$2.924\,100\times10^{10}$	6
5	74	6	10	14.73	22	2117.48	$2.924\,100\times10^{10}$	6
6	90	7	9	14.49	21	2107.34	$2.924\,100\times10^{10}$	7
7	112	4	18	15.60	24	2097.37	$2.924\,100\times10^{10}$	7
8	857	7	738	20.96	27	2088.00	$2.924\,100\times10^{10}$	6
9	969	4	108	18.18	26	2073.74	$2.924\,100\times10^{10}$	8
10	1481	5	507	20.51	28	2075.49	$2.924\,100\times10^{10}$	8
11	1625	4	140	18.61	29	2060.73	$1.728\,000\times10^{9}$	1

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
12	2564	7	932	21.31	28	2046.84	$1.728\,000\times10^9$	10
13	2902	2	336	19.87	27	2042.94	$1.728\,000\times10^9$	7
14	2915	4	9	14.49	20	2037.02	$1.728\,000\times10^9$	13
15	3107	10	182	19.05	26	2016.73	$1.728\,000\times10^9$	8
16	3121	4	10	14.54	23	2018.37	$1.728\,000\times10^9$	9
17	3134	1	12	14.96	22	2013.18	$1.728\,000\times10^9$	10
18	3262	3	125	18.39	28	1995.47	$1.728\,000\times10^9$	9
19	3300	3	35	16.63	26	1983.97	$1.728\,000\times10^9$	7
20	3436	5	131	18.48	28	1980.10	$1.728\,000\times10^9$	2
21	3595	6	153	18.81	29	1963.48	$1.728\,000\times10^9$	3
22	3719	4	120	18.41	28	1959.50	$1.728\,000\times10^9$	3
23	3725	3	3	12.21	27	1943.55	$1.728\,000\times10^9$	5
24	3739	2	12	14.98	26	1930.20	$1.728\,000\times10^9$	4
25	3753	5	9	14.58	28	1916.25	$1.728\,000\times10^9$	7
26	3758	4	1	7.00	27	1913.48	$1.728\,000\times10^9$	2
27	3765	4	3	12.17	26	1901.27	$1.728\,000\times10^9$	3
28	3770	3	2	9.81	27	1891.11	$1.728\,000\times10^9$	10
29	3778	6	2	9.58	26	1853.77	$1.728\,000\times10^9$	7
30	3786	7	1	7.00	25	1824.55	$1.728\,000\times10^9$	7
31	3965	78	101	18.08	25	1130.23	$7.860\,000\times10^8$	18

Table A.23.: Interactive ShortProjectionSR of NTRU lattice
($N = 107$, $a = 0.602$, $c = 1.794$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.90$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
1	31	29	2	11.52	15	2145.43	$3.168\,400\times10^{10}$	6
2	45	5	9	14.48	20	2139.03	$3.168\,400\times10^{10}$	6
3	215	9	161	18.68	24	2120.67	$3.168\,400\times10^{10}$	6
4	228	6	7	13.97	21	2113.66	$3.168\,400\times10^{10}$	7
5	296	10	58	17.20	25	2113.98	$3.168\,400\times10^{10}$	7
6	1176	20	860	21.08	29	2083.96	$2.627\,098\times10^9$	1
7	1192	6	10	14.58	20	2074.25	$2.627\,098\times10^9$	9
8	1250	15	43	16.81	23	2068.68	$2.627\,098\times10^9$	7
9	1263	8	5	13.44	23	2063.57	$2.627\,098\times10^9$	2
10	1295	8	24	15.91	25	2050.33	$2.627\,098\times10^9$	3
11	1307	9	3	12.36	25	2038.17	$2.627\,098\times10^9$	4
12	1337	26	4	12.88	21	2020.77	$2.356\,849\times10^9$	1
13	1347	8	2	9.00	20	2019.45	$2.356\,849\times10^9$	2
14	1352	3	2	8.58	20	2004.49	$2.356\,849\times10^9$	4
15	1362	9	1	7.00	25	1982.63	$2.356\,849\times10^9$	6
16	1371	7	2	7.00	23	1948.32	$2.253\,933\times10^9$	9
17	1512	139	2	7.00	21	1138.26	$7.416\,494\times10^8$	4

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$	e

Table A.24.: Interactive ShortProjectionSR of NTRU lattice
($N = 109$, $a = 0.601$, $c = 1.817$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.90$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
1	31	25	6	13.78	18	2258.24	$3.276\,100\times10^{10}$	6
2	59	25	3	12.67	20	2216.02	$3.276\,100\times10^{10}$	7
3	75	14	2	11.95	18	2213.62	$3.276\,100\times10^{10}$	6
4	95	14	6	13.67	19	2193.44	$3.276\,100\times10^{10}$	6
5	108	10	3	12.67	23	2174.75	$3.276\,100\times10^{10}$	7
6	387	16	263	19.24	26	2163.23	$6.302\,000\times10^{9}$	1
7	705	10	308	19.48	25	2155.95	$6.302\,000\times10^{9}$	7
8	800	11	84	17.65	23	2152.74	$6.302\,000\times10^{9}$	8
9	864	21	43	16.63	22	2142.30	$6.302\,000\times10^{9}$	9
10	881	6	11	14.58	24	2138.24	$6.302\,000\times10^{9}$	9
11	1046	16	149	18.46	27	2122.98	$6.302\,000\times10^{9}$	7
12	1096	21	29	16.11	25	2118.87	$6.302\,000\times10^{9}$	8
13	1110	8	6	13.63	23	2114.00	$6.302\,000\times10^{9}$	9
14	1205	7	88	17.67	30	2098.99	$3.190\,000\times10^{9}$	1
15	1244	4	35	16.30	29	2089.57	$3.190\,000\times10^{9}$	2
16	1261	11	6	13.44	28	2071.35	$3.190\,000\times10^{9}$	3
17	1279	15	3	11.64	26	2052.30	$3.190\,000\times10^{9}$	3
18	1286	5	2	9.58	24	2044.98	$1.734\,000\times10^{9}$	1
19	1295	7	2	10.46	24	2026.04	$1.734\,000\times10^{9}$	5
20	1308	11	2	8.58	24	2025.75	$1.734\,000\times10^{9}$	4
21	1464	154	2	8.00	24	1172.03	$8.120\,000\times10^{8}$	4

Table A.25.: Interactive ShortProjectionSR of NTRU lattice
($N = 109$, $a = 0.602$, $c = 1.789$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.90$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
1	43	42	1	8.58	11	2333.66	$2.220\,000\times10^{8}$	6
2	78	34	1	8.00	14	2314.82	$2.220\,000\times10^{8}$	6
3	104	25	1	10.17	16	2313.54	$2.220\,000\times10^{8}$	6
4	118	13	1	7.00	13	2315.38	$2.220\,000\times10^{8}$	7
5	136	16	2	12.00	16	2272.98	$2.220\,000\times10^{8}$	6
6	149	10	3	12.21	18	2268.83	$2.220\,000\times10^{8}$	7
7	168	14	5	13.44	22	2264.94	$2.220\,000\times10^{8}$	7
8	191	5	18	15.35	22	2253.02	$2.220\,000\times10^{8}$	6
9	221	19	11	14.57	20	2246.27	$2.220\,000\times10^{8}$	7

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
10	323	5	97	17.79	24	2240.23	$2.220\,000\times10^8$	7
11	502	12	167	18.56	26	2235.21	$2.220\,000\times10^8$	6
12	1316	3	811	20.87	27	2228.15	$2.220\,000\times10^8$	7
13	1416	5	95	17.74	30	2229.26	$2.220\,000\times10^8$	6
14	1561	7	138	18.31	28	2207.30	$2.220\,000\times10^8$	2
15	1698	10	127	18.17	27	2200.18	$2.220\,000\times10^8$	8
16	1709	5	6	13.64	33	2188.97	$2.220\,000\times10^8$	3
17	1765	6	50	16.82	32	2178.04	$2.220\,000\times10^8$	2
18	1923	11	147	18.38	31	2166.94	$2.220\,000\times10^8$	5
19	1939	11	5	13.04	30	2149.69	$2.220\,000\times10^8$	4
20	1953	9	5	13.25	29	2138.55	$2.220\,000\times10^8$	3
21	1959	4	2	10.32	28	2135.90	$2.220\,000\times10^8$	4
22	1976	10	7	13.63	27	2117.95	$2.220\,000\times10^8$	4
23	2011	29	6	13.29	27	2090.38	$2.220\,000\times10^8$	8
24	2019	6	2	8.00	25	2066.45	$2.220\,000\times10^8$	10
25	2177	156	2	7.00	24	1228.67	$2.220\,000\times10^8$	3

Table A.26.: Interactive ShortProjectionSR of NTRU lattice
($N = 111$, $a = 0.600$, $c = 1.773$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.90$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
1	18	17	1	9.81	13	2372.39	$3.648\,100\times10^{10}$	8
2	58	39	1	8.00	15	2345.47	$3.648\,100\times10^{10}$	8
3	77	16	3	12.13	17	2340.82	$3.648\,100\times10^{10}$	8
4	140	31	32	16.06	22	2319.27	$3.648\,100\times10^{10}$	8
5	172	25	7	13.77	20	2304.01	$3.648\,100\times10^{10}$	9
6	196	15	9	14.07	21	2302.15	$3.648\,100\times10^{10}$	9
7	208	6	6	13.55	25	2296.52	$3.648\,100\times10^{10}$	8
8	300	15	77	17.39	23	2284.07	$3.648\,100\times10^{10}$	9
9	647	17	330	19.49	31	2273.18	$3.648\,100\times10^{10}$	8
10	676	22	7	13.73	28	2254.74	$3.648\,100\times10^{10}$	9
11	1133	13	444	19.98	27	2257.75	$3.648\,100\times10^{10}$	10
12	1293	6	154	18.38	26	2250.05	$3.648\,100\times10^{10}$	11
13	3004	6	1705	21.93	31	2250.35	$3.648\,100\times10^{10}$	10
14	3034	15	15	14.86	24	2232.29	$3.648\,100\times10^{10}$	13
15	3059	13	12	14.53	25	2226.57	$3.648\,100\times10^{10}$	11
16	3250	8	183	18.70	27	2230.14	$3.648\,100\times10^{10}$	12
18	5091	10	3	9.81	13	2224.74	$3.648\,100\times10^{10}$	16
19	5098	3	4	11.75	16	2223.71	$3.648\,100\times10^{10}$	17
20	5133	8	27	15.80	20	2220.73	$3.648\,100\times10^{10}$	15
21	5150	6	11	14.22	22	2218.85	$3.648\,100\times10^{10}$	17
22	5979	7	822	20.82	28	2212.83	$3.648\,100\times10^{10}$	15

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
23	6093	12	102	17.82	29	2209.11	$3.648\,100\times10^{10}$	13
24	6124	9	22	15.50	27	2210.34	$3.648\,100\times10^{10}$	15
25	6896	13	759	20.70	31	2192.38	$5.278\,285\times10^{9}$	1
26	7002	7	99	17.74	30	2186.68	$2.926\,501\times10^{9}$	1
27	7112	9	101	17.80	30	2173.68	$2.926\,501\times10^{9}$	3
28	7262	12	138	18.24	29	2160.61	$2.926\,501\times10^{9}$	4
29	7327	7	58	16.98	27	2153.88	$2.926\,501\times10^{9}$	12
30	7346	10	9	14.04	26	2143.56	$2.926\,501\times10^{9}$	5
31	7360	5	9	13.83	26	2137.65	$2.926\,501\times10^{9}$	3
32	7385	5	20	15.32	26	2135.04	$2.926\,501\times10^{9}$	4
33	7393	4	4	11.46	26	2123.83	$1.930\,998\times10^{9}$	1
34	7416	7	16	14.94	26	2109.41	$1.930\,998\times10^{9}$	9
35	7439	20	3	10.46	25	2100.11	$1.930\,998\times10^{9}$	9
36	7448	6	3	8.58	24	2075.05	$1.930\,998\times10^{9}$	10
37	7460	9	3	9.81	24	2070.62	$1.930\,998\times10^{9}$	5
38	7612	149	3	8.00	23	1264.96	$1.044\,206\times10^{9}$	5

Table A.27.: Interactive ShortProjectionSR of NTRU lattice
($N = 113$, $a = 0.591$, $c = 1.780$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.90$, $u_{\max} = 22$)

A.4. Knapsack Lattices

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
1	0	0	0	8.00	14	458.56	$6.220\,000\times10^5$	1
2	0	0	0	8.58	13	452.72	$6.220\,000\times10^5$	2
3	0	0	0	8.00	17	452.58	$5.300\,000\times10^5$	1
4	0	0	0	12.75	20	449.21	$5.300\,000\times10^5$	2
5	13	0	13	18.40	25	450.99	$4.800\,000\times10^5$	2
6	13	0	0	10.46	23	449.55	$4.800\,000\times10^5$	3
7	52	0	39	19.86	26	452.31	$4.800\,000\times10^5$	3
8	56	0	4	16.75	22	445.79	$4.800\,000\times10^5$	5
9	57	0	1	14.42	25	443.72	$4.160\,000\times10^5$	1
10	83	0	26	19.21	24	449.44	$4.160\,000\times10^5$	5
11	143	0	60	20.42	29	444.41	$4.160\,000\times10^5$	2
12	146	0	3	16.36	28	445.05	$4.160\,000\times10^5$	3
13	148	0	2	15.98	27	442.51	$4.160\,000\times10^5$	4
14	154	0	6	17.19	25	442.27	$4.160\,000\times10^5$	5
16	338	0	1	15.15	24	437.38	$4.160\,000\times10^5$	6
17	342	0	4	16.50	24	438.49	$4.160\,000\times10^5$	8
18	365	0	23	19.06	29	437.14	$4.160\,000\times10^5$	6
19	383	0	18	18.72	26	440.60	$4.160\,000\times10^5$	8
20	454	0	71	20.63	27	440.51	$4.160\,000\times10^5$	8
21	469	0	15	18.44	24	436.79	$4.160\,000\times10^5$	9
22	487	0	18	18.61	25	435.32	$4.160\,000\times10^5$	10
23	487	0	0	7.00	19	439.15	$4.160\,000\times10^5$	27
24	487	0	0	10.17	18	434.32	$4.160\,000\times10^5$	27
25	487	0	0	8.58	22	432.41	$4.160\,000\times10^5$	22
26	487	0	0	8.00	21	432.93	$4.160\,000\times10^5$	25
27	487	0	0	10.70	22	434.02	$4.160\,000\times10^5$	21
28	487	0	0	10.58	22	431.40	$4.160\,000\times10^5$	27
29	487	0	0	9.81	21	430.97	$4.160\,000\times10^5$	30
30	487	0	0	7.00	25	430.20	$4.160\,000\times10^5$	22
31	487	0	0	9.32	26	431.79	$4.160\,000\times10^5$	24
32	489	0	2	15.44	23	431.91	$4.160\,000\times10^5$	30
33	495	0	6	17.25	27	430.54	$4.160\,000\times10^5$	30
34	510	0	15	18.43	30	431.50	$4.160\,000\times10^5$	26
35	571	0	61	20.41	30	427.38	$4.160\,000\times10^5$	27
36	571	0	0	13.36	25	430.60	$4.160\,000\times10^5$	33
37	571	0	0	13.13	24	430.35	$4.160\,000\times10^5$	39
38	571	0	0	8.58	27	429.44	$4.160\,000\times10^5$	33
39	571	0	0	13.82	26	428.85	$4.160\,000\times10^5$	40
40	571	0	0	13.63	27	428.53	$4.160\,000\times10^5$	38
41	571	0	0	8.58	27	426.04	$4.160\,000\times10^5$	34
42	676	0	105	21.03	33	430.30	$4.160\,000\times10^5$	31

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert\mathbf{b}_j\rVert^2$	e
43	676	0	0	12.21	26	424.39	$4.160\,000\times10^5$	32
44	676	0	0	8.00	26	426.04	$4.160\,000\times10^5$	36
45	676	0	0	11.09	24	426.01	$4.160\,000\times10^5$	37
46	676	0	0	9.32	22	429.38	$4.160\,000\times10^5$	38
47	676	0	0	9.58	24	427.10	$4.160\,000\times10^5$	39
48	676	0	0	13.66	27	424.43	$4.160\,000\times10^5$	40
49	676	0	0	13.51	29	424.65	$4.160\,000\times10^5$	34
50	676	0	0	12.36	29	424.18	$4.160\,000\times10^5$	36
51	676	0	0	13.04	28	422.81	$4.160\,000\times10^5$	37
52	676	0	0	11.64	33	420.62	$9.000\,000\times10^4$	31
53	676	0	0	10.70	31	423.56	$9.000\,000\times10^4$	40
54	677	0	1	15.10	33	422.66	$9.000\,000\times10^4$	37
55	677	0	0	13.02	34	421.24	$9.000\,000\times10^4$	39
56	677	0	0	8.58	33	422.86	$9.000\,000\times10^4$	40

Table A.28.: Interactive ShortProjectionSR of CJLOSS lattice
($n = 100$, density 0.935 $k = 10$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.90$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert\mathbf{b}_j\rVert^2$	e
1	0	0	0	9.58	16	468.09	$6.980\,000\times10^5$	1
2	0	0	0	10.17	15	462.09	$6.500\,000\times10^5$	2
3	0	0	0	13.23	19	453.62	$6.500\,000\times10^5$	3
4	0	0	0	11.64	17	457.66	$6.500\,000\times10^5$	4
5	0	0	0	9.00	15	450.02	$6.500\,000\times10^5$	5
6	0	0	0	8.58	19	451.65	$6.500\,000\times10^5$	4
7	1	0	1	14.67	22	448.90	$6.500\,000\times10^5$	2
8	1	0	0	11.58	20	450.93	$6.500\,000\times10^5$	5
9	2	0	1	15.16	22	448.93	$6.500\,000\times10^5$	5
10	19	0	17	18.59	25	448.38	$6.500\,000\times10^5$	5
11	28	0	9	17.72	28	452.15	$5.080\,000\times10^5$	1
12	29	0	1	15.36	28	447.43	$5.080\,000\times10^5$	4
13	39	0	10	17.78	26	449.43	$5.080\,000\times10^5$	5
14	63	0	24	19.08	29	448.00	$5.080\,000\times10^5$	4
15	93	0	30	19.40	28	446.66	$5.080\,000\times10^5$	5
17	285	0	9	17.70	25	441.19	$5.080\,000\times10^5$	8
18	384	0	99	21.11	27	439.49	$5.080\,000\times10^5$	6
19	384	0	0	10.81	22	439.81	$5.080\,000\times10^5$	10
20	386	0	2	15.82	26	439.65	$5.080\,000\times10^5$	6
21	397	0	11	17.94	28	436.35	$5.080\,000\times10^5$	5
22	415	0	18	18.69	29	437.82	$5.080\,000\times10^5$	10
23	415	0	0	13.36	25	436.65	$5.080\,000\times10^5$	12
24	419	0	4	16.67	28	437.95	$5.080\,000\times10^5$	9

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
25	434	0	15	18.43	28	435.74	$5.080\,000\times10^5$	10
26	434	0	0	9.00	19	435.98	$5.080\,000\times10^5$	27
27	434	0	0	10.32	20	434.06	$5.080\,000\times10^5$	26
28	434	0	0	11.70	21	434.72	$5.080\,000\times10^5$	24
29	434	0	0	8.00	22	428.62	$5.080\,000\times10^5$	26
30	434	0	0	13.02	22	425.54	$9.000\,000\times10^4$	28
31	434	0	0	12.36	26	423.54	$9.000\,000\times10^4$	27
32	434	0	0	14.10	25	426.03	$9.000\,000\times10^4$	30
33	493	0	59	20.33	30	421.54	$9.000\,000\times10^4$	22

Table A.29.: Interactive ShortProjectionSR of CJLOSS lattice
($n = 100$, density 0.833 $k = 10$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.90$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2$ odef(B)	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
1	0	0	0	9.00	15	460.02	$7.225\,000\times10^5$	1
2	0	0	0	9.32	22	453.52	$5.900\,000\times10^5$	1
3	0	0	0	11.52	21	456.48	$5.900\,000\times10^5$	2
4	41	0	41	19.87	28	450.85	$5.295\,000\times10^5$	1
5	41	0	0	9.00	19	453.25	$5.295\,000\times10^5$	4
6	44	0	3	16.09	23	450.16	$5.295\,000\times10^5$	4
7	46	0	2	15.89	22	451.20	$5.295\,000\times10^5$	5
8	46	0	0	10.58	16	446.92	$5.295\,000\times10^5$	7
9	46	0	0	8.00	14	451.32	$5.295\,000\times10^5$	8
10	46	0	0	7.00	16	445.54	$5.295\,000\times10^5$	9
11	46	0	0	13.73	19	449.39	$5.295\,000\times10^5$	8
12	46	0	0	9.00	18	448.36	$5.295\,000\times10^5$	9
13	47	0	1	14.79	24	452.44	$5.295\,000\times10^5$	8
14	50	0	3	16.11	22	445.47	$5.295\,000\times10^5$	9
15	87	0	37	19.66	26	448.63	$5.295\,000\times10^5$	7
16	88	0	1	14.69	25	445.80	$5.295\,000\times10^5$	8
17	90	0	2	15.39	27	449.01	$5.295\,000\times10^5$	8
18	123	0	33	19.53	30	443.85	$5.295\,000\times10^5$	2
19	147	0	24	19.10	26	441.68	$5.295\,000\times10^5$	6
20	152	0	5	16.83	25	438.85	$5.295\,000\times10^5$	8
21	189	0	37	19.73	28	443.18	$5.295\,000\times10^5$	3
22	189	0	0	11.95	20	441.29	$5.295\,000\times10^5$	25
23	189	0	0	9.32	22	443.79	$5.295\,000\times10^5$	30
24	189	0	0	10.32	25	437.48	$5.295\,000\times10^5$	21
25	189	0	0	10.00	21	436.69	$5.295\,000\times10^5$	28
26	189	0	0	10.32	23	433.94	$5.295\,000\times10^5$	29
27	191	0	2	15.67	23	438.09	$5.295\,000\times10^5$	22

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
28	191	0	0	7.00	22	434.52	$5.295\,000\times10^5$	25
29	193	0	2	15.71	24	439.25	$5.295\,000\times10^5$	23
30	193	0	0	10.46	23	437.85	$5.295\,000\times10^5$	30
31	193	0	0	11.64	24	432.96	$5.295\,000\times10^5$	24
32	193	0	0	13.93	23	431.68	$5.295\,000\times10^5$	25
33	193	0	0	12.00	21	430.76	$5.295\,000\times10^5$	26
34	193	0	0	14.00	24	431.57	$5.295\,000\times10^5$	28
35	202	0	9	17.74	27	432.23	$5.295\,000\times10^5$	29
36	202	0	0	10.17	23	430.68	$5.295\,000\times10^5$	37
37	202	0	0	7.00	26	433.53	$5.295\,000\times10^5$	38
38	203	0	1	14.59	31	432.52	$5.295\,000\times10^5$	37
39	203	0	0	13.21	31	431.13	$5.295\,000\times10^5$	34
40	204	0	1	14.86	30	427.92	$5.295\,000\times10^5$	38
41	204	0	0	14.10	31	429.87	$5.295\,000\times10^5$	40
42	205	0	1	15.31	34	429.88	$5.295\,000\times10^5$	40
43	205	0	0	14.34	37	427.10	$5.295\,000\times10^5$	33
45	389	0	0	12.81	31	431.90	$5.295\,000\times10^5$	45
46	390	0	1	14.59	45	428.51	$5.295\,000\times10^5$	47
47	390	0	0	13.19	44	427.65	$5.295\,000\times10^5$	48
49	578	0	0	14.02	43	425.58	$5.295\,000\times10^5$	49
50	578	0	0	10.32	46	426.38	$5.295\,000\times10^5$	50
51	652	0	74	20.67	45	428.24	$5.295\,000\times10^5$	42
52	656	0	4	16.67	25	426.76	$5.000\,000\times10^5$	4
53	669	0	13	18.26	27	427.78	$5.000\,000\times10^5$	5
54	751	0	82	20.85	30	424.83	$4.745\,000\times10^5$	2
55	752	0	1	14.42	24	428.28	$4.745\,000\times10^5$	8
56	760	0	8	17.58	25	424.52	$4.745\,000\times10^5$	8
57	761	0	1	14.23	24	425.51	$4.745\,000\times10^5$	10
58	765	0	4	16.54	28	425.29	$4.745\,000\times10^5$	9
59	768	0	3	16.36	27	425.94	$4.745\,000\times10^5$	10
60	777	0	9	17.62	29	429.58	$4.745\,000\times10^5$	4
61	781	0	4	16.63	26	425.57	$4.745\,000\times10^5$	10
62	798	0	17	18.59	30	431.37	$4.745\,000\times10^5$	9
63	798	0	0	11.25	31	423.54	$4.745\,000\times10^5$	5
64	826	0	28	19.29	29	421.27	$4.745\,000\times10^5$	6
65	837	0	11	18.03	27	422.15	$4.745\,000\times10^5$	7
66	839	0	2	15.82	26	424.00	$4.745\,000\times10^5$	6
67	843	0	4	16.67	26	422.93	$4.745\,000\times10^5$	9
68	844	0	1	14.51	24	425.10	$4.745\,000\times10^5$	10
69	852	0	8	17.47	27	424.92	$4.745\,000\times10^5$	8
70	865	0	13	18.17	28	418.32	$4.745\,000\times10^5$	9
71	865	0	0	11.17	24	425.28	$4.745\,000\times10^5$	26
72	865	0	0	10.46	24	421.97	$4.745\,000\times10^5$	29

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
73	865	0	0	13.61	23	422.79	$4.745\,000\times10^5$	23
74	865	0	0	13.39	25	424.22	$4.745\,000\times10^5$	21
75	865	0	0	11.25	24	422.16	$4.745\,000\times10^5$	28
76	866	0	1	15.11	28	420.29	$4.745\,000\times10^5$	28
77	868	0	2	16.01	27	420.83	$4.745\,000\times10^5$	25
78	868	0	0	13.49	25	420.57	$4.745\,000\times10^5$	30
79	874	0	6	17.06	28	417.45	$4.745\,000\times10^5$	24
80	874	0	0	13.11	26	419.63	$4.745\,000\times10^5$	29
81	876	0	2	15.91	26	417.56	$4.745\,000\times10^5$	25
82	876	0	0	7.00	26	421.36	$4.745\,000\times10^5$	26
83	878	0	2	15.61	27	422.37	$4.745\,000\times10^5$	27
84	878	0	0	12.43	25	418.32	$4.745\,000\times10^5$	33
85	878	0	0	11.09	23	417.25	$4.745\,000\times10^5$	34
86	878	0	0	7.00	26	415.59	$4.745\,000\times10^5$	37
87	878	0	0	10.58	26	417.48	$4.745\,000\times10^5$	39
88	880	0	2	15.59	31	416.07	$4.745\,000\times10^5$	33
89	881	0	1	15.04	30	418.88	$4.745\,000\times10^5$	40
90	883	0	2	15.65	32	418.00	$4.745\,000\times10^5$	34
91	883	0	0	9.32	30	414.63	$4.745\,000\times10^5$	35
92	883	0	0	14.01	28	418.05	$4.745\,000\times10^5$	36
93	883	0	0	13.11	27	419.24	$4.745\,000\times10^5$	37
94	885	0	2	16.01	34	415.98	$4.745\,000\times10^5$	40
96	1069	0	2	16.05	36	417.19	$4.745\,000\times10^5$	43
97	1069	0	0	13.21	37	416.96	$4.745\,000\times10^5$	44
98	1069	0	0	11.09	34	415.94	$4.745\,000\times10^5$	45
101	1265	0	0	11.70	30	414.88	$4.745\,000\times10^5$	44
102	1265	0	0	9.58	30	416.29	$4.745\,000\times10^5$	45
103	1265	0	0	13.30	30	418.24	$4.745\,000\times10^5$	39
104	1265	0	0	11.64	29	417.01	$4.745\,000\times10^5$	40
105	1265	0	0	8.00	31	417.76	$4.745\,000\times10^5$	41
106	1265	0	0	11.39	30	417.23	$4.745\,000\times10^5$	42
107	1265	0	0	10.58	30	415.89	$4.745\,000\times10^5$	43
108	1265	0	0	8.00	30	416.23	$4.745\,000\times10^5$	44
109	1265	0	0	9.32	28	414.38	$4.745\,000\times10^5$	44
110	1265	0	0	14.13	37	416.24	$4.745\,000\times10^5$	45
111	1265	0	0	9.00	36	414.82	$4.745\,000\times10^5$	46
112	1265	0	0	7.00	33	415.66	$4.745\,000\times10^5$	47
113	1265	0	0	8.00	34	416.20	$4.745\,000\times10^5$	49
114	1265	0	0	8.58	45	417.49	$4.745\,000\times10^5$	50
115	1265	0	0	8.00	39	417.66	$4.745\,000\times10^5$	48
116	1265	0	0	8.58	41	412.88	$4.745\,000\times10^5$	49
117	1265	0	0	7.00	43	416.58	$4.745\,000\times10^5$	50
118	1265	0	0	13.02	52	416.97	$4.745\,000\times10^5$	45

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
119	1268	0	3	16.37	54	416.32	$4.745\,000\times10^5$	48
120	1268	0	0	14.34	53	413.29	$4.745\,000\times10^5$	50
122	1457	0	0	12.09	1000	415.61	$4.745\,000\times10^5$	54
123	1457	0	0	8.00	1000	416.52	$4.745\,000\times10^5$	59
124	1457	0	0	8.58	1000	416.13	$4.745\,000\times10^5$	55
125	1457	0	0	10.91	1000	415.96	$4.745\,000\times10^5$	51
126	1457	0	0	11.09	1000	416.28	$4.745\,000\times10^5$	57
127	1462	0	5	17.00	1000	416.33	$4.745\,000\times10^5$	60
129	1650	0	5	17.02	1000	414.06	$4.745\,000\times10^5$	51
130	1652	0	2	16.03	1000	414.03	$4.745\,000\times10^5$	51
131	1657	0	5	16.85	27	415.98	$4.745\,000\times10^5$	7
132	1657	0	0	11.86	25	410.80	$4.745\,000\times10^5$	8
133	1657	0	0	10.91	24	416.90	$4.745\,000\times10^5$	9
134	1706	0	49	20.08	28	415.59	$4.745\,000\times10^5$	10
135	1706	0	0	11.95	30	416.36	$4.745\,000\times10^5$	9
136	1717	0	11	18.02	28	417.95	$4.745\,000\times10^5$	10
138	1906	0	0	7.00	21	415.30	$4.745\,000\times10^5$	17
139	1906	0	0	13.46	21	413.90	$4.745\,000\times10^5$	19
140	1906	0	0	11.17	20	416.04	$4.745\,000\times10^5$	20
141	1907	0	1	14.41	22	412.97	$4.745\,000\times10^5$	13
142	1907	0	0	10.70	24	412.69	$4.745\,000\times10^5$	20
143	1907	0	0	7.00	24	416.69	$4.745\,000\times10^5$	19
144	1907	0	0	11.17	24	411.94	$1.275\,000\times10^5$	1

Table A.30.: Interactive ShortProjectionSR of CJLOSS lattice
($n = 100$, density 0.833 $k = 15$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.90$, $u_{\max} = 22$)

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
1	0	0	0	8.00	13	444.17	$5.860\,000\times10^5$	1
2	0	0	0	7.00	18	447.17	$4.900\,000\times10^5$	1
3	0	0	0	8.00	22	443.26	$4.140\,000\times10^5$	1
4	0	0	0	10.70	16	445.47	$4.140\,000\times10^5$	3
5	1	0	1	14.91	21	440.24	$4.140\,000\times10^5$	2
6	10	0	9	17.76	23	438.31	$4.140\,000\times10^5$	4
7	10	0	0	13.44	22	438.06	$4.140\,000\times10^5$	5
8	25	0	15	18.45	25	435.62	$4.100\,000\times10^5$	4
9	26	0	1	14.55	23	437.21	$4.100\,000\times10^5$	4
10	162	0	136	21.60	29	440.02	$4.100\,000\times10^5$	5
11	166	0	4	16.44	23	439.70	$4.100\,000\times10^5$	7
12	169	0	3	16.37	23	435.96	$4.100\,000\times10^5$	8
13	174	0	5	16.87	28	438.46	$4.100\,000\times10^5$	7
14	174	0	0	13.82	22	439.51	$4.100\,000\times10^5$	10
15	175	0	1	14.73	27	433.96	$4.100\,000\times10^5$	8

Continued on next page

iter	t_{total}	t_{BKZ}	t_{sample}	$\log_2$ #samples	$u_{\min}$	$\log_2 \operatorname{odef}(\mathsf{B})$	$\min_j \lVert \mathbf{b}_j \rVert^2$	e
16	177	0	2	15.60	26	441.02	$4.100\,000\times10^5$	9
17	192	0	15	18.39	27	436.72	$4.100\,000\times10^5$	10
18	299	0	107	21.17	31	438.52	$4.100\,000\times10^5$	7
19	317	0	18	18.56	25	434.34	$4.100\,000\times10^5$	12
20	320	0	3	16.25	24	437.37	$4.100\,000\times10^5$	13
21	324	0	4	16.36	23	433.00	$4.100\,000\times10^5$	8
22	326	0	2	15.64	29	433.73	$4.100\,000\times10^5$	9
23	365	0	39	19.76	28	430.62	$4.100\,000\times10^5$	13
24	370	0	5	16.77	27	431.91	$4.100\,000\times10^5$	14
25	370	0	0	9.81	21	430.63	$4.100\,000\times10^5$	30
26	370	0	0	13.60	22	429.37	$4.100\,000\times10^5$	23
27	370	0	0	11.25	22	427.01	$4.100\,000\times10^5$	27
28	370	0	0	8.00	22	430.42	$4.100\,000\times10^5$	28
29	370	0	0	12.13	20	427.86	$4.100\,000\times10^5$	29
30	371	0	1	14.38	25	427.54	$4.100\,000\times10^5$	30
31	387	0	16	18.36	29	425.77	$4.100\,000\times10^5$	23
32	387	0	0	13.07	24	427.50	$4.100\,000\times10^5$	32
33	387	0	0	12.52	25	426.79	$4.100\,000\times10^5$	35
34	387	0	0	11.09	25	426.65	$4.100\,000\times10^5$	37
35	388	0	1	15.01	27	424.63	$4.100\,000\times10^5$	38
36	395	0	7	17.33	28	421.77	$4.100\,000\times10^5$	39
38	583	0	0	8.58	41	425.50	$4.100\,000\times10^5$	45
39	583	0	0	11.39	49	421.64	$4.100\,000\times10^5$	49
40	583	0	0	13.38	48	424.32	$4.100\,000\times10^5$	50
42	769	0	0	9.32	34	422.41	$4.100\,000\times10^5$	48
43	769	0	0	8.00	34	423.04	$4.100\,000\times10^5$	50
44	769	0	0	7.00	36	423.82	$4.100\,000\times10^5$	44
45	769	0	0	8.00	31	422.76	$4.100\,000\times10^5$	45
46	769	0	0	8.58	36	421.76	$4.100\,000\times10^5$	46
47	769	0	0	11.64	38	420.16	$4.100\,000\times10^5$	41
48	769	0	0	7.00	34	421.25	$4.100\,000\times10^5$	50
49	769	0	0	7.00	35	422.16	$4.100\,000\times10^5$	41
50	769	0	0	8.00	31	422.52	$4.100\,000\times10^5$	46
52	981	0	22	18.91	28	422.39	$4.100\,000\times10^5$	5
53	1004	0	23	19.01	30	421.08	$9.000\,000\times10^4$	1

Table A.31.: Interactive ShortProjectionSR of CJLOSS lattice
($n = 100$, density 0.980 $k = 10$, $\delta = 0.99$, $\beta = 5$, $\gamma = 0.90$, $u_{\max} = 22$)

Bibliography

[ABSS97] Arora, S., Babai, L., Stern, J., and Sweedyk, E. Z. *The Hardness of Approximate Optima in Lattices, Codes, and Systems of Linear Equations.* Journal of Computer and System Sciences, volume 54, no. 2: pages 317–331, 1997.

[AEVZ02] Agrell, E., Eriksson, T., Vardy, A., and Zeger, K. *Closest Point Search in Lattices.* IEEE Transactions on Information Theory, volume 48, no. 8: pages 2201–2214, 2002.

[AG04] Abrahams, D. and Gurtovoy, A. *C++ Template Metaprogramming.* Addison-Wesley, Boston, 2004.

[Ajt98] Ajtai, M. *The Shortest Vector Problem in L_2 is NP-hard for Randomized Reductions (Extended Abstract).* In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, pages 10–19. ACM Press, 1998. ISBN 0-89791-962-9.

[Ajt03] Ajtai, M. *The Worst-case Behavior of Schnorr's Algorithm Approximating the Shortest Nonzero Vector in a Lattice.* In: Larmore and Goemans [LG03], pages 396–406.

[AKS01] Ajtai, M., Kumar, R., and Sivakumar, D. *A Sieve Algorithm for the Shortest Vector Problem.* In: J. S. Vitter, P. Spirakis, and M. Yannakakis (editors), *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing.* ACM Press, 2001.

[Ale00] Alexandrescu, A. *Traits: The else-if-then of Types.* C++ Report, April 2000.

[Ale01] Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, Boston, 2001. ISBN 0-201-70431-5.

[AR03] Aharonov, D. and Regev, O. *A Lattice Problem in Quantum NP.* In: *Proc. 44th Annual IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 210–219. 2003.

[AR04] Aharonov, D. and Regev, O. *Lattice Problems in NP intersect coNP.* In: *Proc. 45th Annual IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 362–371. 2004.

[Bab86] Babai, L. *On Lovász' Lattice Reduction and the Nearest Lattice Point Problem.* Combinatorica, volume 6, no. 1: pages 1–13, 1986.

[BBHT96] Boyer, M., Brassard, G., Høyer, P., and Tapp, A. *Tight Bounds on Quantum Searching.* arXiv e-print quant-ph/9605034, 1996.

[BCD+04] Buchmann, J., Coronado, C., Döring, M., Engelbert, D., Ludwig, C., Overbeck, R., Schmidt, A., Vollmer, U., and Weinmann, R.-P. *Post-Quantum Signatures.* Cryptology ePrint Archive, Report 2004/297, 2004. URL http://eprint.iacr.org/.

[Bel00] Bellare, M. (editor). *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *LNCS*. Springer, 2000.

[BHT98] Brassard, G., Høyer, P., and Tapp, A. *Quantum Counting.* arXiv e-print quant-ph/9805082, 1998.

[BKLW99] Blochinger, W., Küchlin, W., Ludwig, C., and Weber, A. *An Object-Oriented Platform for Distributed High-Performance Symbolic Computation.* Mathematics and Computers in Simulation, volume 49: pages 161 – 178, 1999.

[BO91] Brickell, E. F. and Odlyzko, A. M. *Cryptanalysis: A Survey of Recent Results.* In: G. J. Simmons (editor), *Contemporary Cryptology*, pages 578–593. IEEE Press, 1991.

[Boo04] *Boost – Peer Reviewed Portable C++ Source Libraries*, Nov. 2004. URL http://www.boost.org/. Release 1.32.0.

[BS89] Bronštejn, I. N. and Semendjaev, K. A. *Taschenbuch der Mathematik.* BSB Teubner, Leipzig, 24th edition, 1989. ISBN 3-322-00259-4.

[Buc94] Buchmann, J. *Reducing Lattice Bases by Means of Approximations.* In: L. M. Adleman (editor), *Algorithmic Number Theory (ANTS I)*, volume 877 of *LNCS*, pages 160–168. Springer, 1994.

[CJL+92] Coster, M. J., Joux, A., LaMacchia, B. A., Odlyzko, A. M., Schnorr, C.-P., and Stern, J. *Improved low-density subset sum algorithms.* Comput. Complexity, volume 2, no. 2: pages 111–128, 1992.

[Coh96] Cohen, H. *A Course in Computational Algebraic Number Theory.* Number 138 in Graduate Texts in Mathematics. Springer-Verlag, Berlin, Heidelberg, New York, 3rd edition, 1996.

[CS97] Coppersmith, D. and Shamir, A. *Lattice Attacks on NTRU.* In: *Advances in Cryptology – Eurocrypt'97*, volume 1233 of *LNCS*, pages 52–61. Springer, 1997.

[DKRS03] Dinur, I., Kindler, G., Raz, R., and Safra, S. *Approximating CVP to within Almost-Polynomial Factors is NP-hard*. Combinatorica, volume 23, no. 2: pages 205–243, 2003.

[EB81] Emde Boas, P. v. *Another NP-Complete Partition Problem and the Complexity of Computing Short Vectors in a Lattice*. Technical Report 81–04, University of Amsterdam, Department of Mathematics, Netherlands, 1981.

[FJ03] Frigo, M. and Johnson, S. G. *FFTW Version 3.0.1 – User Manual*, June 2003. URL http://www.fftw.org.

[FJ05] Frigo, M. and Johnson, S. G. *The Design and Implementation of FFTW3*. Proceedings of the IEEE, volume 93, no. 2: pages 216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[FK90] Fischer, H. and Kaul, H. *Mathematik für Physiker, Band 1*. B. G. Teubner, Stuttgart, 2nd edition, 1990.

[For84] Forster, O. *Analysis 3 – Integralrechnung im $\mathbb{R}^n$ mit Anwendungen*. Vieweg, Braunschweig, 3rd edition, 1984. ISBN 3-528-27252-X.

[Gau01] Gauß, C. F. *Disquisitiones Arithmeticae*. Leipzig, 1801.

[GG00] Goldreich, O. and Goldwasser, S. *On the Limits of Nonapproximability of Lattice Problems*. Journal of Computing and System Sciences, volume 60, no. 3: pages 540–563, 2000.

[GGH97] Goldreich, O., Goldwasser, S., and Halevi, S. *Public-Key Cryptosystems from Lattice Reduction Problems*. In: B. S. Kaliski, Jr. (editor), *Advances in Cryptology – Crypto'97*, volume 1294 of *LNCS*, pages 112–131. Springer-Verlag, 1997.

[GHJV95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, 1995. ISBN 0-201-63361-2.

[GJSS01] Gentry, C., Jonsson, J., Stern, J., and Szydlo, M. *Cryptanalysis of the NTRU Signature Scheme (NSS) from Eurocrypt 2001*. In: C. Boyd (editor), *Advances in Cryptology – Asiacrypt 2001*, volume 2248 of *LNCS*, pages 1–20. Springer-Verlag, 2001.

[GL96] Golub, G. H. and van Loan, C. F. *Matrix Computations*. Johns Hopkins University Press, 3rd edition, 1996.

[Gle05] Glendinning, I. *Quantum Computing Projects and Funding*, Aug. 2005. URL http://www.vcpc.univie.ac.at/~ian/hotlist/qc/projects.shtml.

[Gro96] Grover, L. K. *A Fast Quantum Mechanical Algorithm for Database Search*. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing (STOC)*, pages 212–219. ACM Press, 1996. ISBN 0-89791-785-5.

[GS02] Gentry, C. and Szydlo, M. *Cryptanalysis of the Revised NTRU Signature Scheme*. In: L. Knudsen (editor), *Advances in Cryptology – Eurocrypt 2002*, volume 2332 of *Lecture Notes in Copmuter Science*, pages 299–320. Springer-Verlag, 2002.

[Hec95] Heckler, C. *Automatische Parallelisierung und parallele Gitterbasisreduktion*. Ph.D. thesis, Universität des Saarlandes, 1995.

[HGNP+03] Howgrave-Graham, N., Nguyen, P. Q., Pointcheval, D., Proos, J., Silverman, J. H., Singer, A., and Whyte, W. *The Impact of Decryption Failures on the Security of NTRU Encryption*. In: D. Boneh (editor), *Advances in Cryptology - CRYPTO 2003*, volume 2729/2003 of *LNCS*, pages 226–246. Springer, 2003.

[HGSW03] Howgrave-Graham, N., Silverman, J. H., and Whyte, W. *A Meet-In-The-Middle Attack on an NTRU Private Key*. Technical Report 4, version 2, NTRU Cryptosystems, 2003. URL http://www.ntru.com/cryptolab/pdf/NTRUTech004v2.pdf.

[HHGP+02] Hoffstein, J., Howgrave-Graham, N., Pipher, J., Silverman, J. H., and Whyte, W. *NTRUSign: Digital Signatures Using the NTRU Lattice*. Technical Report, NTRU Cryptosystems, Inc., April 2002. URL http://www.ntru.com/cryptolab/pdf/NTRUSign-preV2.pdf. Preliminary Draft.

[HHGP+03] Hoffstein, J., Howgrave-Graham, N., Pipher, J., Silverman, J., and Whyte, W. *NTRUSign: Digital Signatures Using the NTRU Lattice*. In: M. Joye (editor), *Topics in Cryptology – CT-RSA 2003*, volume 2612 of *LNCS*, pages 122–140. Springer, 2003.

[HHGP+05] Hoffstein, J., Howgrave-Graham, N., Pipher, J., Silverman, J. H., and Whyte, W. *Performance Improvements and a Baseline Parameter Generation Algorithm for NTRUSign*. Technical Report, NTRU Inc., 2005. URL http://www.ntru.com/cryptolab/pdf/NTRUSignParams-2005-08.pdf.

[HL95] Hürsch, W. L. and Lopes, C. V. *Separation of Concerns*. Technical Report, College of Computer Science, Northeastern University, Boston, 1995. URL ftp://ftp.ccs.neu.edu/pub/people/lieber/crista/techrep95/separation.pdf.

[HPS98] Hoffstein, J., Pipher, J., and Silverman, J. H. *NTRU: A Ring-Based Public Key Cryptosystem*. In: J. P. Buhler (editor), *Algorithmic Number Theory (ANTS III)*, volume 1423 of *LNCS*. Springer-Verlag, 1998.

[HPS01] Hoffstein, J., Pipher, J., and Silverman, J. H. *NSS: An NTRU Lattice-Based Signature Scheme*. In: B. Pfitzmann (editor), *Advances in Cryptology – Eurocrypt 2001*, volume 2045 of *LNCS*, pages 211–228. Springer-Verlag, 2001.

[HS00] Hoffstein, J. and Silverman, J. H. *Optimizations for NTRU*. In: *Public-Key Cryptography and Computational Number Theory, Warsaw, Sep 11-15, 2000*. 2000. URL http://www.ntru.com/cryptolab/articles.htm#002.

[HSW03] Hoffstein, J., Silverman, J. H., and Whyte, W. *Estimated Breaking Times for NTRU Lattices*. Technical Report 12, version 2, NTRU Cryptosystems, 2003. URL http://www.ntru.com/cryptolab/tech_notes.htm#012.

[JJ00] Jaulmes, É. and Joux, A. *A Chosen-Ciphertext Attack against NTRU*. In: Bellare [Bel00], pages 20–35.

[Kan83] Kannan, R. *Improved Algorithms for Integer Programming and Related Lattice Problems*. In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 193–206. ACM Press, 1983.

[Kan87] Kannan, R. *Minkowski's Convex Body Theorem and Integer Programming*. Math. Oper. Research, volume 12: pages 415–440, 1987.

[Koy04] Koy, H. *Primale/duale Segment-Reduktion von Gitterbasen*. Slides of a talk (in German), 2004. URL http://www.mi.informatik.uni-frankfurt.de/research/papers/primdual.ps.

[KS01a] Koy, H. and Schnorr, C. P. *Segment LLL-Reduction of Lattice Bases*. In: Silverman [Sil01], pages 67–80.

[KS01b] Koy, H. and Schnorr, C. P. *Segment LLL-Reduction with Floating Point Orthogonalization*. In: Silverman [Sil01], pages 81–96.

[KS02] Koy, H. and Schnorr, C. P. *Segment and Strong Segment LLL-Reduction of Lattice Bases*, 2002. URL http://www.mi.informatik.uni-frankfurt.de/research/papers.html. Preprint.

[Len83] Lenstra, H. W., Jr. *Integer Programming with a Fixed Number of Variables*. Mathematics Operations Research, volume 8: pages 538–548, 1983.

[LG03] Larmore, L. L. and Goemans, M. X. (editors). *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. ACM Press, 2003. ISBN 1-58113-674-9.

[LG04] LiDIA-Group. LiDIA – *A Library for Computational Number Theory*. TU Darmstadt, 2004. URL http://www.informatik.tu-darmstadt.de/TI/LiDIA/Welcome.html. Release 2.1.3.

[Lis88] Liskov, B. *Data Abstraction and Hierarchy*. SIGPLAN Not., volume 23, no. 5: pages 17–34, 1988. ISSN 0362-1340.

[LLL82] Lenstra, A. K., Lenstra, H. W., and Lovász, L. *Factoring Polynomials with Rational Coefficients*. Math. Ann., volume 261: pages 515–534, 1982.

[LLS90] Lagarias, C. J., Lenstra, H. W., and Schnorr, C. *Korkine-Zolotarev Bases and Successive Minima of a Lattice and its Reciprocal Lattice*. Combinatorica, volume 10, no. 4: pages 333–348, 1990.

[LO85] Lagarias, J. C. and Odlyzko, A. M. *Solving Low-Density Subset Sum Problems*. J. Assoc. Comput. Mach., volume 32: pages 229–246, 1985.

[Lud02] Ludwig, C. *The Security and Efficiency of Micciancio's Cryptosystem*. Technical Report TI-7/02, TU Darmstadt, Germany, 2002. URL http://www.informatik.tu-darmstadt.de/ftp/pub/TI/TR/TI-02-07.MiccPaper.pdf.

[Lud03] Ludwig, C. *A Faster Lattice Reduction Method Using Quantum Search*. In: T. Ibaraki, N. Katoh, and H. Ono (editors), *ISAAC 2003: 14th International Symposium on Algorithms and Computations*, volume 2906 of *LNCS*, pages 199–208. Springer, 2003.

[LV01] Lenstra, A. K. and Verheul, E. R. *Selecting Cryptographic Key Sizes*. J. Cryptology, volume 14, no. 4: pages 255–293, 2001.

[MG02] Micciancio, D. and Goldwasser, S. *Complexity of Lattice Problems*. Kluwer Academic Publishers, 2002. ISBN 0-7923-7688-9.

[Mic01a] Micciancio, D. *The Hardness of the Closest Vector Problem with Preprocessing*. IEEE Transactions on Information Theory, volume 47, no. 3: pages 1212–1215, 2001.

[Mic01b] Micciancio, D. *Improving Lattice Based Cryptosystems Using the Hermite Normal Form*. In: Silverman [Sil01], pages 126–145.

[Mic01c] Micciancio, D. *The Shortest Vector Problem is NP-hard to approximate to Within Some Constant*. SIAM Journal on Computing, volume 30, no. 6: pages 2008–2035, 2001.

[MN98] Matsumoto, M. and Nishimura, T. *Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator*. ACM Trans. Model. Comput. Simul., volume 8, no. 1: pages 3–30, 1998. ISSN 1049-3301.

[MS01] May, A. and Silverman, J. H. *Dimension Reduction Methods for Convolution Modular Lattices*. In: Silverman [Sil01], pages 110–125.

[MYK04] Min, S., Yamamoto, G., and Kim, K. *Weak Property of Malleability in NTRUSign*. In: *ACISP04*. 2004.

[NC00] Nielsen, M. A. and Chuang, I. L. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[Ngu99] Nguyen, P. *Cryptanalysis of the Goldreich-Goldwasser-Halevi Cryptosystem from Crypto'97*. In: M. Wiener (editor), *Advances in Cryptology – Crypto'99*, volume 1666 of *LNCS*, pages 288–304. Springer-Verlag, 1999.

[NP02] Nguyen, P. Q. and Pointcheval, D. *Analysis and Improvements of NTRU Encryption Paddings*. In: M. Yung (editor), *Advances in Cryptology - CRYPTO 2002*, volume 2442 of *LNCS*, pages 210–225. Springer-Verlag, 2002.

[NS01] Nguyen, P. Q. and Stern, J. *The Two Faces of Lattices in Cryptography*. In: Silverman [Sil01], pages 146–180.

[Odl90] Odlyzko, A. M. *The Rise and Fall of Knapsack Crytosystems*. In: *Cryptology and Computational Number Theory*, volume 42 of *Proc. Symp. Appl. Math.*, pages 75–88. AMS, 1990.

[OT03] Oomura, K. and Tanaka, K. *Density Attack on the Knapsack Cryptosystems with Enumerative Source Encoding (Extended Abstract)*, February 2003. URL `citeseer.ist.psu.edu/615682.html`.

[OTU00] Okamoto, T., Tanaka, K., and Uchiyama, S. *Quantum Public-Key Cryptosystems*. In: Bellare [Bel00], pages 147–165.

[PTVF92] Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1992. ISBN 0-521-43108-5.

[Pyt05] Python Software Foundation. *Python*, 2005. URL `http://www.python.org/`. Release 2.4.1.

[Reg02] Regev, O. *Quantum Computations and Lattice Problems*. In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS'02)*, pages 520–529. IEEE, 2002.

[Reg03] Regev, O. *New Lattice Based Cryptographic Constructions*. In: Larmore and Goemans [LG03], pages 407–416.

[Reg05] Regev, O. *On Lattices, Learning with Errors, Random Linear Codes, and Cryptography*. In: *Proc. 37th ACM Symp. on Theory of Computing (STOC)*, pages 84–93. 2005.

[Sch87] Schnorr, C. P. *A Hierachy of Polynomial Lattice Basis Reduction Algorithms*. Theoretical Computer Science, volume 53: pages 201–224, 1987.

[Sch94] Schnorr, C. P. *Block Reduced Lattice Bases and Successive Minima*. Comb., Prob., and Comp., volume 3: pages 507–522, 1994.

[Sch03] Schnorr, C. P. *Lattice Reduction by Random Sampling and Birthday Methods*. In: H. Alt and M. Habib (editors), *STACS 2003: 20th Annual Symposium on Theoretical Aspects of Computer Science*, volume 2607 of *LNCS*, pages 146–156. Springer, 2003.

[Sci04] *SciPy – Scientific Tools for Python*, 2004. URL http://www.scipy.org/.

[SE94] Schnorr, C. P. and Euchner, M. *Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems*. Math. Programming, volume 66: pages 181–199, 1994.

[Sho94] Shor, P. W. *Algorithms for Quantum Computation: Discrete Logarithms and Factoring*. In: *IEEE Symposium on Foundations of Computer Science*, pages 124–134. 1994.

[Sho97] Shor, P. W. *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*. SIAM J. Comput., volume 26, no. 5: pages 1484–1509, 1997.

[Sho05] Shoup, V. *NTL – A Library for Doing Number Theory*, 2005. URL http://www.shoup.net/ntl/index.html. Release 5.4.

[Sil01] Silverman, J. H. (editor). *Cryptography and Lattices*, volume 2146 of *LNCS*. Springer-Verlag, 2001.

[Unr02] Unruh, E. *Template Metaprogrammierung*, 2002. URL http://www.erwin-unruh.de/meta.html.

[Wei05] Weisstein, E. W. *Fresnel Integrals*. From *MathWorld* – a Wolfram Web Resource., June 2005. URL http://mathworld.wolfram.com/FresnelIntegrals.html.

[Wet98] Wetzel, G. S. *Lattice Basis Reduction Algorithms and their Applications*. Ph.D. thesis, Universität des Saarlandes, Saarbrücken, 1998.