Towards a Modern LLL Implementation

Léo Ducas[1,2], Ludo N. Pulles[1], and Marc Stevens[1]

[1] CWI, Cryptology Group, Amsterdam, the Netherlands
[2] Mathematical Institute, Leiden University, Leiden, the Netherlands

Abstract. We propose BLASter, a proof of concept LLL implementation that demonstrates the practicality of multiple theoretical improvements. The implementation uses the segmentation strategy from Neumaier–Stehlé (ISSAC 2016), parallelism and Seysen's reduction that was proposed by Kirchner–Espitau–Fouque (CRYPTO 2021) and implemented in OptLLL, and the BLAS library for linear algebra operations. It consists of only 1000 significant lines of C++ and Python code, and is made publicly available.

For q-ary lattices that fplll can handle without multiprecision (dimension < 180), BLASter is considerably faster than fplll, OptLLL and Ryan–Heninger's flatter (CRYPTO 2023), without degrading output reduction quality. Thanks to Seysen's reduction it can further handle larger dimension without resorting to multiprecision, making it more than 10x faster than flatter and OptLLL, and 100x faster than fplll in dimensions 256 to 1024.

It further includes segmented BKZ and segmented deep-LLL variants. The latter provides bases as good as BKZ-15 and has a runtime that is only a couple of times more than our LLL baseline.

This remains a proof of concept: the effective use of higher precision — which is needed to handle *all* lattices — has further obstacles and is left for future work. Still, this work contains many lessons learned, and is meant to motivate and guide the development of a robust and modern lattice reduction library, which shall be much faster than fplll.

Keywords: Lattice reduction · LLL · Segmentation · Parallelism · Concrete performance · Divide and Conquer

1 Introduction

Implementing the LLL algorithm [39] is a notoriously hard task. A testimony of this claim is the limited progress between 2005 and 2019, despite various asymptotic improvements proposed in the literature [57,50,45]. This stands in stark contrast with the other lattice algorithms such as enumeration [24,14] and sieving [4,17], for which we have up-to-date software [68,2].

Finally in 2019, Kirchner, Espitau and Fouque proposed a faster reduction library OptLLL [35]. This got followed-up by Ryan and Heninger with a more robust one, flatter [53]. Still, there remain some documented ideas that are left unimplemented up to now. Furthermore, flatter is still based on fplll [68], whose

very design dates from a different computational era. Modern computer architectures offer massive vectorization and parallelization, which are very suitable to speed up linear algebra tasks.

Early applications of lattice reduction construct lattice bases with possibly very large entries [39,38,9,10,15,31], motivating a line of research, lowering the runtime dependence on the entry sizes [34,55,47,50], of which [47] is implemented in fplll. More recent cryptosystems [62,40,51,8] base their security on the hardness of solving a lattice problem [32,52] given a basis having completely different characteristics: very high dimension and *very small entries*. This motivates us to focus on q-ary lattices with a small q.

In this regime, the runtime dependence on the dimension is more important and most linear algebra can be computed with machine word precision, which can be delegated to modern optimized BLAS libraries. Although fplll and flatter use machine word precision until numerical issues are encountered, they hardly depend on BLAS.

To improve the runtime dependence on the dimension, Schönhage started a line of work to let lattice reduction run recursively and repeatedly within segments until the basis is sufficiently reduced [60,66,36,37,57,45,35,53]. Although initial works provided a weaker reduction notion than LLL, Neumaier and Stehlé construct an algorithm outputting a basis, of which the first vector has a norm comparable to LLL [45]. The algorithm relies on fast integer multiplication algorithms, and has a runtime complexity of $\mathcal{O}(n^{4+\epsilon}\log(B)^{1+\epsilon})$ for any $\epsilon > 0$, where n is the dimension and B bounds the norm of all basis vectors. This algorithm has never been implemented, and remains the fastest provable lattice reduction algorithm thus far.

It is therefore quite plausible that LLL can still be made much faster, but this requires starting from scratch. Admittedly, a fast and robust lattice reduction library is a huge project, because it would require multiprecision computations that do not degrade performance. In particular, an important missing tool is an optimized BLAS library that effectively can handle multiprecision data. Currently, OptLLL and flatter use a naïve solution with dynamically allocated multiprecision elements inside matrices, which prevents the effective use of massive vectorization, parallelization and CPU cache optimizations of modern optimized BLAS libraries.

1.1 Related Work

Kirchner–Espitau–Fouque [35]. Kirchner, Espitau and Fouque give a *heuristic* lattice reduction algorithm OptLLL with claimed runtime $\mathcal{O}(n^\omega C)$, where C is the logarithm of the condition number of the input basis, and ω the matrix multiplication exponent. They use Seysen's reduction instead of size reduction. This algorithm uses a strong heuristic, namely that the *Geometric Series Assumption* (GSA) [56] holds throughout execution, and may crash or not terminate when a lattice violates this heuristic [53, Sec. 6.3 & 6.5].

The algorithm uses a segmented recursive approach similar to [45], and suggests that it is possible to reduce disjoint segments in parallel, using an idea

dating back to Villard [69,28]. However, it does not discuss the optimal number of segments at any depth, when using multithreading. Moreover, when running the software, all but one thread are idle most of the time.

Their asymptotic runtime is optimistic in practice, because it uses both fast integer multiplication and matrix multiplication algorithms that are infeasible in practice. For example, the Coppersmith–Winograd algorithm [16] achieves $\omega \leq 2.376$, but is admittedly slower than the naive algorithm for matrices of dimension $n \leq 10,000$ [29]. Moreover, the GMP library prefers FFT-based integer multiplication rather than Toom–Cook variants such as [7] for integers of 3000 machine words or more.[1] As a consequence, in practice it may not be optimal to use Schönhage's algorithm for rank 2 lattice reduction [61], and their proposed blockwise Cholesky decomposition.

The OptLLL software can be downloaded at the bottom of Espitau's webpage `https://espitau.github.io/fastlll.html`.

Ryan–Heninger [53]. Ryan and Heninger propose both a provable algorithm and a heuristic algorithm to reduce lattices, called flatter. The heuristic algorithm has a time complexity of $\mathcal{O}(n^\omega(C+n)^{1+\epsilon})$. The algorithm uses ideas of basis compression [54] to reduce the basis to one with bounded so-called *drop*, which is incomparable but similar to the guarantees of LLL.

The algorithm uses a recursive strategy with half-overlapping blocks of size $n/2$, similar to [45]. Utilizing multithreading, flatter reduces the first half and last half in parallel. Moreover, the matrix multiplication implementation utilizes multiple threads.

Interestingly, during QR decomposition, Seysen's reduction is used, but it does not replace size reduction.

If flatter encounters a projected sublattice of rank at most 32 with a needed bit precision of less than 120, it calls fplll to run LLL or BKZ with block size 10, 20, 28 or 35, depending on the desired root Hermite factor for this local basis.

Despite usage of the Eigen library, flatter merely uses the QR decomposition with `doubles` [26], and has self-written algorithms for QR decomposition of MPFRs, and for matrix multiplication.

1.2 Our Contribution

In this work, we propose BLASter, a fast and modern implementation of LLL, which serves as a proof of concept. Namely, our work focuses on reducing lattices that can be handled with machine precision. By algorithmically making use of the segmentation strategy of Neumaier–Stehlé [45], we can handle q-ary lattices comfortably in dimensions up to 1000 with at most 30 bits. In addition, we replace size reduction by Seysen's faster reduction strategy [63], and we directly run QR decomposition instead of a block Cholesky decomposition [35]. Multithreading is extensively used for linear algebra, and for running LLL in parallel

[1] See: `https://gmplib.org/devel/thres/MUL_FFT_THRESHOLD`

on disjoint segments. Moreover, we provide a detailed description for the design choices that were made in the implementation.

Additionally, BLASter contains a variant of deep-LLL [59], BLASterDeepLLL, and a variant of BKZ [59], BLASterBKZ. The BKZ variant preprocesses the basis with BLASterDeepLLL of depth $d = 4$, and then uses a progressive strategy from block size 40 onwards.

We benchmark our proof of concept, and other lattice reduction algorithms, on small lattices that fplll can handle with machine words, and lattices of dimension $128, 256, 512$ and 1024. BLASter is an order of magnitude faster than fplll, OptLLL and flatter. For dimensions above 200, our software is more than 100x faster than fplll. In all cases, we observe that our software reduces lattices faster than fplll, OptLLL and flatter. Despite loose worst-case runtime bounds, the experiments show that BLASterDeepLLL is only about 2x slower than the plain LLL-like algorithm in dimensions up to 256, and only a factor 5 slower than our LLL algorithm in dimension 512, while its output quality is similar to BKZ-15 in practice. BLASterBKZ reduces a lattice of dimension 256 with 20-bit entries using one tour of BKZ-60 in less than 2 minutes, while fplll requires more than 5 minutes to LLL-reduce such lattice.

Library	Lines of code	Direct dependencies
fplll [68]	$16,445^{\dagger}$	GMP [25], MPFR [20], JSON-lib
OptLLL [35]	$1,950$	GMP, MPFR, OpenBLAS, FFTW [21]
flatter [53]	$11,155$	GMP, MPFR, OpenBLAS, fplll, Eigen [26]
BLASter (this work)	939	NumPy, Eigen

Table 1. Lines of code of various lattice reduction libraries, excluding comments and blank lines. Numbers were calculated using the tool `cloc`, which is available at `https://github.com/aldanial/cloc`.
† Excludes precomputed data for the default pruning strategy.

The lattice basis reduction software, BLASter, is publicly available on GitHub:

`https://github.com/ludopulles/BLASter`

The code consists of only *1000 lines of code*, and is written in `Python` (40%), `C++` (45%) and `Cython` (15%) [5]. It relies on NumPy and Eigen [26] for linear algebra. Conveniently, these operations are multithreaded and well-optimized. Namely, NumPy selects the most performant BLAS library available during installation, and Eigen uses OpenMP by default. Table 1 compares the lines of code in our work with other libraries, and lists their dependencies.

Although our code is publicly available, we certainly advise *not to integrate* our limited and fragile library BLASter anywhere. Still, the implementation offers many lessons learned, and acts as motivation towards a modern and robust LLL implementation. We believe such an implementation can be much faster than

state of the art, even when using sufficient precision to work on all lattices. We suggest specific directions for future works in our conclusions (Section 6).

2 Preliminaries

Notation. The natural logarithm (base e) is denoted by log and the binary logarithm (base 2) is denoted by lg.

Matrices and vectors. We use column vectors throughout the paper, and internally in the software. Only the software's input/output format uses row vectors to be compatible with fplll's and NTL's [64] input/output format.

Matrices and vectors are written in boldface. For a vector $\boldsymbol{x}$, we denote $\mathrm{diag}(\boldsymbol{x})$ for the $n \times n$ diagonal matrix with $\boldsymbol{x}$ on the diagonal in that order. The $n \times n$ identity matrix is denoted by id_n. Given a matrix $\boldsymbol{A}$, we write $\boldsymbol{A}_{[a,b]\times[c,d]}$ for the *contiguous submatrix of* $\boldsymbol{A}$ that consists of all the $\boldsymbol{A}_{i,j}$ with $a \leq i \leq b$ and $c \leq j \leq d$.

The max norm of a matrix $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ is $\|\boldsymbol{A}\|_{\max} = \max_{i,j} |\boldsymbol{A}_{ij}|$. The ℓ_2-norm of a vector $\boldsymbol{x}$ is denoted by $\|\boldsymbol{x}\|$, and the *Frobenius norm* of a matrix $\boldsymbol{A} \in \mathbb{R}^{n\times m}$ is the ℓ_2-norm of $\boldsymbol{A}$ seen as one long vector of length nm, and is denoted by $\|\boldsymbol{A}\|_{\mathrm{F}}$. The *condition number* of a nonsingular matrix $\boldsymbol{A}$ is $\kappa(\boldsymbol{A}) = \|\boldsymbol{A}\|_{\mathrm{F}} \cdot \left\|\boldsymbol{A}^{-1}\right\|_{\mathrm{F}}$.

The *matrix multiplication exponent*, denoted by $\omega \in (2, 3]$, is the smallest value such that two $n \times n$ matrices can be multiplied in time $\mathcal{O}(n^\omega)$. While Strassen showed $\omega < 2.81$ [67], it is known $\omega < 2.38$ [16].

Geometry. The n-dimensional *unit ball* is $\mathcal{B}^n$. The *volume* of any n-dimensional measurable set S is $\mathrm{Vol}_n(S)$.

2.1 Lattices

A *lattice* $\Lambda \subset \mathbb{R}^d$ is a discrete subgroup of $\mathbb{R}^d$.

The $\mathbb{R}$*-linear span of a set* $S \subseteq \mathbb{R}^d$ is denoted by $\mathrm{span}\,(S)$. The *rank of* Λ is the $\mathbb{R}$-linear dimension of $\mathrm{span}\,(\Lambda)$, and we say Λ is *full-rank* if it is of rank d.

The *volume of* Λ, denoted by $\det(\Lambda)$, is given by $\mathrm{Vol}_n(\mathrm{span}\,(\Lambda)\,/\Lambda)$.

A vector $\boldsymbol{v} \in \mathbb{R}^n$ is called *primitive w.r.t.* Λ if $\forall t \in \mathbb{R}\colon t\boldsymbol{v} \in \Lambda \iff t \in \mathbb{Z}$.

The norm of a shortest nonzero vector of a lattice $\Lambda \subset \mathbb{R}^d$ is denoted by $\lambda_1(\Lambda)$. For $q \in \mathbb{Z}_{>1}$, a lattice $\Lambda \subset \mathbb{R}^d$ is *q-ary* if $q\mathbb{Z}^d \subset \Lambda \subset \mathbb{Z}^d$.

For all $n \in \mathbb{Z}_{\geq 1}$ let

$$\mathrm{GH}(n) = \mathrm{Vol}_n(\mathcal{B}^n)^{-1/n} = \frac{\left(\Gamma(\frac{n}{2}+1)\right)^{1/n}}{\sqrt{\pi}}.$$

Any lattice $\Lambda \subset \mathbb{R}^d$ of rank n satisfies $\lambda_1(\Lambda) \leq 2\mathrm{GH}(n)\det(\Lambda)^{1/n}$ due to Minkowski's theorem [42]. The *Gaussian Heuristic* states that a random lattice, for example a random q-ary lattice, has $\lambda_1(\Lambda) \approx \mathrm{GH}(n)\det(\Lambda)^{1/n}$.

Lattice Bases. A *basis for a lattice* $\Lambda \subset \mathbb{R}^d$ is a matrix $\boldsymbol{B} \in \mathbb{R}^{d \times n}$, consisting of $\mathbb{R}$-linearly independent column vectors $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_n \in \mathbb{R}^d$, such that $\Lambda = \mathbb{Z}\boldsymbol{b}_1 + \cdots + \mathbb{Z}\boldsymbol{b}_n$. Note, given a basis $\boldsymbol{B}$ for Λ, we have $\det(\Lambda) = \sqrt{\det(\boldsymbol{B}^\mathsf{T}\boldsymbol{B})}$.

Given a basis $\boldsymbol{B}$, *the fundamental region* $\mathcal{P}(\boldsymbol{B})$ is defined by

$$\mathcal{P}(\boldsymbol{B}) = \left\{ \boldsymbol{x} \in \mathbb{R}^d \;\middle|\; \exists \boldsymbol{c} \in [-1/2,\ 1/2)^n \colon \boldsymbol{x} = \boldsymbol{B}\boldsymbol{c} \right\}.$$

Moreover, the *Babai domain of* $\boldsymbol{B}$ is defined by $\mathcal{P}(\boldsymbol{B}^*)$, where $\boldsymbol{B}^*$ is the Gram–Schmidt orthogonalization of $\boldsymbol{B}$ (see below).

2.2 Lattice Basis Reduction

We assume the reader is familiar with lattice basis reduction. This can be found in the course by Micciancio [41]. For the average behaviour of LLL, see [46]. Additionally, there is a book on LLL [48], containing a chapter on floating point LLL [65], and one on Householder reflections and segments [58].

Gram–Schmidt orthogonalization. Given $\ell \in \{1, 2, \ldots, n\}$ and a basis $\boldsymbol{B} \in \mathbb{R}^{d \times n}$, the projection orthogonally away from the basis vectors $\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{\ell-1}$ is denoted by π_ℓ. For $r \in \{\ell, \ldots, n\}$, we write $\boldsymbol{B}_{[\ell,r]}$ for the basis consisting of the vectors $\pi_\ell(\boldsymbol{b}_\ell), \ldots, \pi_\ell(\boldsymbol{b}_r)$, which is a basis for a *projected sublattice* of Λ.

The *Gram–Schmidt vectors* $\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*$ are given by $\mathbf{b}_i^* = \pi_i(\boldsymbol{b}_i)$. The *Gram–Schmidt coefficients* μ_{ij} are given by $\mu_{ij} = \langle \boldsymbol{b}_j, \mathbf{b}_i^* \rangle / \|\mathbf{b}_i^*\|^2$ for $1 \leq i \leq j \leq n$ (so $\mu_{ii} = 1$). It is well-known that $\det(\Lambda) = \prod_{i=1}^n \|\mathbf{b}_i^*\|$. The *Gram–Schmidt Orthogonalization (GSO)* of $\boldsymbol{B}$ is the pair $(\boldsymbol{d}, \boldsymbol{M})$ where $\boldsymbol{d}_i = \|\mathbf{b}_i^*\|$, and $\boldsymbol{M}_{ij} = \mu_{ij}$ for $i \leq j$ and zero otherwise.

QR decomposition. Given a basis $\boldsymbol{B} \in \mathbb{R}^{d \times n}$, there exists a unique decomposition $\boldsymbol{B} = \boldsymbol{Q}\boldsymbol{R}$ with $\boldsymbol{Q} \in \mathbb{R}^{d \times n}$ and $\boldsymbol{R} \subset \mathbb{R}^{n \times n}$, such that $\boldsymbol{R}$ is an upper triangular matrix with strictly positive diagonal, and $\boldsymbol{Q}^\mathsf{T}\boldsymbol{Q} = \mathrm{id}_n$. The matrix $\boldsymbol{R}$ is called the *R-factor* of a basis $\boldsymbol{B}$. Note that libraries, such as LAPACK and NumPy, do not enforce $\boldsymbol{R}$ to have a positive diagonal.

The QR decomposition and GSO orthogonalization can both be computed using at most $\mathcal{O}(dn^2)$ floating point operations. Householder reflections can be used to compute the QR decomposition or only the R-factor [30, Chapter 19].

The original L^2 paper [47] does not use Householder reflections, because its floating point accuracy was not understood at the time, even though it was known in practice that Householder reflections are numerically more stable than standard Gram–Schmidt orthogonalization [37,57]. The perturbation analysis of the R-factor in [11] was instrumental in proving correctness of the "H-LLL" algorithm [43], which uses Householder reflections to compute the QR decomposition. This algorithm has been implemented in fplll thereafter.

Note that $\boldsymbol{R} = \mathrm{diag}(\boldsymbol{d})\boldsymbol{M}$, i.e. $\boldsymbol{R}_{ii} = \|\mathbf{b}_i^*\|$ and $\boldsymbol{R}_{ij} = \boldsymbol{d}_i \mu_{ij}$ $(1 \leq i \leq j \leq n)$.

Size reduction. A basis $\boldsymbol{B}$ is *size-reduced* if its GSO $(\boldsymbol{d}, \boldsymbol{M})$ of $\boldsymbol{B}$ satisfies

$$\forall i, j \colon (i < j) \implies |\mu_{ij}| \leq \frac{1}{2}.$$

Babai's nearest plane (NP) algorithm [3] is an algorithm that receives as input a basis $\boldsymbol{B} \in \mathbb{R}^{d \times n}$ and a "target" $\boldsymbol{t} \in \mathbb{R}^d$, outputs a vector $\boldsymbol{c} \in \mathbb{Z}^n$, such that $\boldsymbol{t} + \boldsymbol{B}\boldsymbol{c} \in \mathcal{P}(\boldsymbol{B}^*)$ holds. It is well-known that NP requires $\mathcal{O}(dn)$ floating point operations.

The usual way to perform size reduction is to reduce $\boldsymbol{b}_i$ by calling NP with $[\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{i-1}]$ and $\boldsymbol{b}_i$ where i ranges from 2 to n. Thus, size reduction can be done using $\mathcal{O}(dn^2)$ floating point operations.

Seysen's reduction. Seysen's reduction [63], similar to size reduction, controls the size of *both* the primal and dual vectors in terms of their respective Gram–Schmidt vectors. Applying Seysen's reduction to the R-factor instead of naïve size reduction improves its condition number from exponential to quasi-polynomial, and therefore allows running LLL with significantly lower precision [35,53,12].

The following is a recursive definition in terms of n, which was originally only defined for even n [63, Proposition 5].

Definition 1 (Seysen's reduction). *A basis $\boldsymbol{B} \in \mathbb{R}^{d \times n}$ is called* Seysen-reduced *if* either $n = 1$ or *its R-factor $\boldsymbol{R} \in \mathbb{R}^{n \times n}$, parsed as*

$$\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix},$$

with $\boldsymbol{R}_{11}$ a square matrix of order $\lfloor n/2 \rfloor$, satisfies these three properties: $\boldsymbol{R}_{11}$ is Seysen-reduced, $\boldsymbol{R}_{22}$ is Seysen-reduced, and $\left\| \boldsymbol{R}_{11}^{-1} \boldsymbol{R}_{12} \right\|_{\max} \leq \frac{1}{2}$.

Algorithm 1 SeysenReduce($\boldsymbol{R}$)

1: **if** $\boldsymbol{R} \in \mathbb{R}^{1 \times 1}$ **then**
2: **return** $\begin{bmatrix} 1 \end{bmatrix}$
3: Parse $\begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix} \leftarrow \boldsymbol{R}$ ▷ Blocks have half the size
4: $\boldsymbol{U}_{11} \leftarrow$ SeysenReduce($\boldsymbol{R}_{11}$)
5: $\boldsymbol{U}_{22} \leftarrow$ SeysenReduce($\boldsymbol{R}_{22}$)
6: $\boldsymbol{R}_{12} \leftarrow \boldsymbol{R}_{12}\boldsymbol{U}_{22}$
7: $\boldsymbol{U}'_{12} \leftarrow \lfloor -\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12} \rceil$
8: $\boldsymbol{R}_{12} \leftarrow \boldsymbol{R}_{11}\boldsymbol{U}'_{12} + \boldsymbol{R}_{12}$ ▷ $\boldsymbol{R}_{11}$ is Seysen-reduced by Line 4
9: **return** $\begin{bmatrix} \boldsymbol{U}_{11} & \boldsymbol{U}_{11}\boldsymbol{U}'_{12} \\ \boldsymbol{0} & \boldsymbol{U}_{22} \end{bmatrix}$

The definition automatically leads to an algorithm that upon input an upper triangular matrix $\boldsymbol{R} \in \mathbb{R}^{n \times n}$ outputs a unimodular matrix $\boldsymbol{U} \in \mathbb{Z}^{n \times n}$ such that

$\boldsymbol{RU}$ is upper triangular and Seysen-reduced [63,35]. One can easily implement the algorithm [63,35] such that $\boldsymbol{R}$ gets Seysen-reduced *in situ*, which we include in Algorithm 1 for later reference. Namely, when Algorithm 1 has received $\boldsymbol{R}$ as input, and returns $\boldsymbol{U}$ as output, the value of $\boldsymbol{R}$ is changed to, say $\boldsymbol{S}$, such that $\boldsymbol{S} = \boldsymbol{RU}$ *and* $\boldsymbol{S}$ is Seysen-reduced. Although less trivial, it is also possible to implement the algorithm without recursion.

Because inverting an upper triangular $n \times n$-matrix takes $\mathcal{O}(n^\omega)$ floating point operations [35, Lemma 1], Seysen's reduction requires $\mathcal{O}(n^\omega)$ floating point operations [35, Theorem 2]. Closer inspection reveals that the practical runtime is dominated by four matrix multiplications and one matrix inversion of size $n/2$.

Remark 1. Both [35] and [53] use size reduction and Seysen's reduction as synonyms, but we use size reduction in its strict sense of $\frac{1}{2}$-size reduction [65].

LLL reduction. Given a $\delta \in (\frac{1}{4}, 1)$, a basis $\boldsymbol{B}$ is called *δ-weakly LLL-reduced* if it satisfies *Lovász' condition*:

$$\forall i \in [1, n-1] : \ \delta \boldsymbol{R}_{ii}^2 \leq \boldsymbol{R}_{i,i+1}^2 + \boldsymbol{R}_{i+1,i+1}^2. \tag{1}$$

A basis is *δ-LLL-reduced* if it is both size-reduced and δ-weakly LLL-reduced.

Given a depth $d \in [1, n]$, a basis is *d-deep δ-LLL-reduced* if

$$\forall i < j : \ (i \leq d \vee i \geq j - d) \implies \delta \boldsymbol{R}_{ii}^2 \leq \sum_{k=i}^{j} \boldsymbol{R}_{kj}^2, \tag{2}$$

which is stronger than Equation (1).

Remark 2. It can be shown that Seysen's reduction and size reduction give the same *superdiagonal*, i.e. $(\boldsymbol{R}_{i,i+1})_{i=1}^{n-1}$. Namely, one can check Seysen's reduction yields $|\boldsymbol{R}_{m,m+1}| \leq \frac{1}{2} |\boldsymbol{R}_{m,m}|$, where $m = \lfloor n/2 \rfloor$, and all other entries follow by induction. Thus, Lovász' condition holds after size reduction iff it holds after Seysen's reduction.

2.3 Basis Reduction Quality

The *basis profile* of a basis $\boldsymbol{B} \in \mathbb{R}^{d \times n}$ is $(\lg \|\mathbf{b}_i^*\|)_{i=1}^n$. There are three ways to quantify the quality of a basis $\boldsymbol{B}$:

1. The *n-th root Hermite factor (RHF)* is given by $\left(\|\boldsymbol{b}_1\| / \det(\Lambda)^{1/n} \right)^{1/n}$. The RHF is sometimes denoted by δ in the literature.
2. The *slope*, denoted by $\mathrm{sl}(\boldsymbol{B})$, is the slope of the line given by simple linear regression on the graph of the basis profile. Explicitly,

$$\mathrm{sl}(\boldsymbol{B}) = \frac{\sum_{i=1}^n \lg \|\mathbf{b}_i^*\| \left(i - \frac{n+1}{2}\right)}{\sum_{i=1}^n \left(i - \frac{n+1}{2}\right)^2}.$$

3. The *potential* is given by:

$$\mathrm{Pot}(\boldsymbol{B}) = \prod_{i=1}^{n} \|\mathbf{b}_i^*\|^{2(n-i+1)} = \prod_{i=1}^{n} \det(\boldsymbol{B}_{[1,i]})^2.$$

If $\boldsymbol{B}^\mathsf{T}\boldsymbol{B} \in \mathbb{Z}^{n\times n}$, the potential is integral [39, (1.25)]. When LLL performs a swap, the potential is reduced by some constant fraction [39], but that is not necessarily the case when reducing with deep-LLL or BKZ. Note that the potential and the slope are related as follows:

$$\lg \mathrm{Pot}(\boldsymbol{B}) + \binom{n+1}{3} \mathrm{sl}(\boldsymbol{B}) = (n+1) \lg \det \Lambda.$$

The goal of lattice basis reduction is to decrease the potential of a basis, or equivalently to *flatten the basis profile*, i.e. make the slope less negative.

During lattice reduction, most iterations will modify the slope but not the RHF, because $\boldsymbol{b}_1$ is seldomly replaced by a shorter vector. Thus, it is more insightful to track the evolution of the slope during lattice reduction.

3 The BLASter LLL Algorithm

Algorithm 2 $\mathsf{BLASter}(\boldsymbol{B}, \delta, \ell)$

Require: $\delta \in (\frac{1}{4}, 1)$, and ℓ is even

$(\boldsymbol{U}, i_0) \leftarrow (\boldsymbol{I}_n, 0)$
repeat
  $\boldsymbol{R} \leftarrow \mathsf{QRDecompose}(\boldsymbol{B})$
  $i_0 \leftarrow \ell/2 - i_0$
  $\mathcal{I} \leftarrow \{\ (i_0 + k\ell + 1, \min(n, i_0 + k\ell + \ell)) \mid 0 \le k < (n - i_0)/\ell\ \}$
  for all $(i,j) \in \mathcal{I}$ **do in parallel**
    $\boldsymbol{V}^{(i)} \leftarrow \mathsf{LLLReduce}(\boldsymbol{R}_{[i,j]\times[i,j]}, \delta)$
  for all $(i,j) \in \mathcal{I}$ **do**
    $\begin{bmatrix}\boldsymbol{B}_{[1,d]\times[i,j]} \\ \boldsymbol{U}_{[1,n]\times[i,j]}\end{bmatrix} \leftarrow \begin{bmatrix}\boldsymbol{B}_{[1,d]\times[i,j]} \\ \boldsymbol{U}_{[1,n]\times[i,j]}\end{bmatrix} \cdot \boldsymbol{V}^{(i)}$
  $\boldsymbol{R} \leftarrow \mathsf{QRDecompose}(\boldsymbol{B})$
  $\boldsymbol{W} \leftarrow \mathsf{SeysenReduce}(\boldsymbol{R})$
  $\begin{bmatrix}\boldsymbol{B} \\ \boldsymbol{U}\end{bmatrix} \leftarrow \begin{bmatrix}\boldsymbol{B} \\ \boldsymbol{U}\end{bmatrix} \cdot \boldsymbol{W}$
until $\boldsymbol{R}$ is δ-weakly LLL-reduced
return $(\boldsymbol{B}, \boldsymbol{U})$

In this section, we give a detailed description of the proposed parallel, segmented LLL-variant, called BLASter. See Algorithm 2 for a high-level overview of our proposed algorithm. It is reminiscent of the algorithm `RecursiveReduce`

by Neumaier and Stehlé [45], and uses a parallel reduction similar to [35, Alg. 5], which is a block version of [69]. Contrary to the recursive designs [45,35,53] with arbitrary depth, Algorithm 2 uses precisely $r = 3$ recursive layers with the following block sizes: $n_1 = 2, n_2 = \ell$ and $n_3 = n$, where we reuse notation from [45]. In particular, we refer to the top level n_3 as the *global* basis, and refer to the middle level n_2 as the *local* blocks or segments.

Contrary to [45]'s "rolling-pin" strategy that rolls backwards, our algorithm leverages multithreading by reducing disjoint blocks of size ℓ concurrently, and if we sort all segments by starting index, then during odd iterations, we reduce the odd segments, and during even iterations the even ones, which is rather different from the "rolling-pin" strategy of moving backwards [45].

Moreover, we deviate from [45,35,53]: the global basis gets Seysen-reduced using matrix operation, but the local segments get size-reduced element-wise. The rationale for this distinction is that when working on local blocks, we need to continuously update the basis, but each update acts only on a few vectors; applying matrix operations would be wasteful. On the contrary, when we return to the global basis, essentially each vector of each segment has been modified, and it is therefore worth resorting to matrix operation implemented in BLAS software.

Remark 3. Note that the algorithm works on lattices of any rank. Moreover, the algorithm can be trivially modified to also work on Gram matrices, $\boldsymbol{G} = \boldsymbol{B}^\mathsf{T}\boldsymbol{B}$, by replacing the QR decomposition of $\boldsymbol{B}$ by the Cholesky decomposition of $\boldsymbol{G}$, and updating $\boldsymbol{G}$ from left and right: $\boldsymbol{G} \leftarrow \boldsymbol{W}^\mathsf{T}\boldsymbol{G}\boldsymbol{W}$.

If a basis $\boldsymbol{B}$ generates a lattice of low rank, say $n < d/2$, one could consider reducing its $n \times n$ Gram matrix instead, because this removes the embedding dimension d, and the Gram matrix only needs to computed once. The transformation matrix that is output, can then be applied to $\boldsymbol{B}$.

Below, we explain in detail the design choices of various aspects of this algorithm and our implementation thereof.

Line 3: QR decomposition. We make use of the function `numpy.linalg.qr` to compute the R-factor of a basis $\boldsymbol{B}$, which a BLAS library usually implements with Householder reflections. We implemented a variant of the blockwise Cholesky decomposition algorithm [35, Algorithm 6] having asymptotic complexity $\mathcal{O}(dn^{\omega-1})$, which can be found in Appendix B. However, in practice this algorithm was much slower than calling QR decomposition. This should not be too surprising: if blockwise Cholesky decomposition would be faster, a well-optimized BLAS library would have used it.

Line 7: LLL reduction on a block. We use a segmented LLL approach to speed up the reduction algorithm. Each segment works on a submatrix of $\boldsymbol{R}$ of dimension at most $\ell \times \ell$.

Our LLLReduce algorithm modifies the block *in situ*, so before calling LLLReduce, a local copy of $\boldsymbol{R}^{(i)} = \boldsymbol{R}_{[i,j]\times[i,j]}$ is made that is *consecutive in memory.*

The local segment only uses two $\ell \times \ell$ matrices: $\boldsymbol{R}^{(i)}$ and $\boldsymbol{V}^{(i)}$, and may fit in a CPU cache if ℓ is not too large.

Moreover, the LLLReduce algorithm locally uses size reduction, and closely follows the pseudocode of Algorithm 1 in [69], an adaptation of the original LLL algorithm [39]. When Lovász' condition is not satisfied at a position k, the kth and $(k+1)$th column of $\boldsymbol{R}^{(i)}$, and $\boldsymbol{V}^{(i)}$ are swapped, a Givens rotation [30, Chapter 19] is performed on $\boldsymbol{R}^{(i)}$.

Note that Schönhage's algorithm [61] is asymptotically more efficient than Lagrange reduction to reduce rank 2 lattices having large integer entries. However, our algorithm simply performs a lightweight swap because we use machine word precision.

Because all the segments are disjoint, all the calls to LLLReduce can be performed concurrently. We discuss this in more detail in Section 3.1.

Line 10: QR decomposition. Remark that we recompute the QR decomposition of $\boldsymbol{B}$ after it was updated with the local LLL transformation matrices $\boldsymbol{V}^{(i)}$. Although $\boldsymbol{R}$ can be multiplied with the $\boldsymbol{V}^{(i)}$ on the right locally, transforming it back to an upper triangular matrix would propagate more rounding errors. Therefore, computing the QR decomposition from $\boldsymbol{B}$ again is much more numerically stable, and it does not significantly impact performance.

Line 11: Size reduction. Mainly for performance reasons, we perform Seysen's reduction on Line 11, which causes the algorithm to only return a δ-weakly LLL-reduced basis.

If one desires to have a δ-LLL-reduced basis, one could size reduce the basis just once before Line 14, which gives a negligible overhead in practice.

Alternatively, one can use *size reduction* as a drop-in replacement for Seysen's reduction. By Remark 2, this replacement does not affect Lovász' condition, so size reduction leads to the same number of iterations.

A naïve size reduction algorithm reduces an R-factor in $\mathcal{O}(n^3)$ time, and does not use matrix operations. However, this algorithm is suboptimal both theoretically and practically. In Appendix A, we derive a size reduction algorithm that performs $\mathcal{O}(n^\omega)$ floating point operations using matrix operations, which we could not find in the literature. Being interested in concrete performance, this means we can use BLAS libraries for size reduction.

Line 14: Return value. Algorithm 2 reduces the basis $\boldsymbol{B}$ *in situ*. Moreover, Algorithm 2, when called with input $\boldsymbol{B}^0$, has the following loop invariant:

$$\boldsymbol{B} = \boldsymbol{B}^0 \cdot \boldsymbol{U}.$$

Thus, Line 14 returns a pair $(\boldsymbol{B}, \boldsymbol{U})$, such that $\boldsymbol{B}$ is δ-weakly LLL-reduced, and $\boldsymbol{B} = \boldsymbol{B}^0 \cdot \boldsymbol{U}$ holds.

3.1 Multithreading

Algorithm 2 makes use of multithreading on multiple CPU cores for the local LLL reductions, and for the matrix operations.

LLL reduction. Line 5 splits the global basis in $\lceil n/\ell \rceil$ smaller disjoint blocks of size ℓ. During the odd iterations (first, third, etc.), these blocks start with index $\ell/2$ modulo ℓ, while the even iterations split the global basis into segments with starting index 0 modulo ℓ.

The time to perform the local lattice reductions in parallel is determined by the slowest segment. Thus, it is best to use $\lceil n/\ell \rceil$ CPU cores, although it is also possible to use fewer. The local LLL reductions cannot make use of more cores than this number, because the local LLL reduction is single-threaded.

When reducing a q-ary lattice, there may be local segments that are either of the form $q\,\mathrm{id}_\ell$ or id_ℓ, which are already LLL-reduced. Those threads that reduce these lattices will terminate instantly, and thus we cannot expect a 100% CPU usage on all cores.

Matrix operations. Lattice reduction makes heavy use of linear algebra. If a linear algebra library uses multithreading, Algorithm 2 can make use of it for QR decomposition (Lines 3 and 10), updating the basis (Lines 9 and 12), and Seysen's reduction (Line 11).

However, note that *too many* threads will slow down matrix operations because this requires creating threads, and moving more memory around.

Local basis updates. Line 9 updates the matrices $\boldsymbol{R}$ and $\boldsymbol{U}$ only in the columns $[i, j]$. These updates are done sequentially by iterating over all $(i, j) \in \mathcal{I}$, but use multithreading for the matrix multiplication. This way, the number of used threads for parallel LLL and matrix operations is constant throughout the execution of the whole program.

Seysen's reduction. Naïvely, one would think Seysen could easily be sped up by computing Lines 4 and 5 in Algorithm 1 in parallel, with each one having half of the number of threads. When these two recursive calls are finished, the remaining matrix operations can be computed with all available threads.

However, this requires controlling carefully the number of threads available for each call, and one needs to wait for both recursive calls to finish, before continuing. These two reasons make this approach impractical in practice.

Alternatively, the matrix operations on the global level dominate the total runtime, while the base cases in total have negligible runtime. Thus, we use all the threads for matrix multiplication. Moreover, note our implementation uses an iterative version of Seysen's reduction in Python, to save on the cost of its function calls.

3.2 Runtime Complexity

In this section, we roughly determine the practical runtime complexity of Algorithm 2, to determine a decent value for ℓ to use in practice. This means we assume matrix multiplication requires n^3 operations, because Strassen's algorithm [67] only gets slightly faster than this naïve algorithm when $n > 500$ [6].

Thus, it is readily seen that one iteration of Algorithm 2 requires the following number of floating point operations:

– $2dn^2$ to run QR decomposition on Lines 3 and 10,
– $(n/\ell)(d+n)\ell^2$ for the local basis updates on Line 9,
– n^3 for Seysen's reduction on Line 11,
– and $(d+n)n^2$ for the global basis update on Line 12.

Note that the cost of Line 9 is at most $2dn\ell$, which is negligible compared to $2dn^2$ when using a small segment size ℓ. Hence, the total cost is $\mathcal{O}(dn^2)$.

Let us now estimate the number of arithmetic operations for the local LLL reduction of a segment, although at any point a floating point may cause the algorithm to loop forever. Since the local LLL is a naïve floating-point LLL (in small dimension though), it performs at most $\mathcal{O}(\ell^4 \lg \max(\boldsymbol{d}))$ arithmetic operations [59], where $\boldsymbol{d}$ is the diagonal of the local R-factor. Note this upper bound is worst-case, because a local segment of a q-ary lattice is sometimes of the form $q\,\mathrm{id}_\ell$ or id_ℓ, which is already LLL-reduced so requires at most ℓ^2 operations.

The number of iterations necessary to satisfy Lovász' condition can be bounded from above using a dynamical system analysis [27,44]. As our algorithm is similar to OptLLL, using [35, Section 3.5] the number of iterations is bounded by $\mathcal{O}\left(\frac{n^2}{\ell^2}\lg\kappa(\boldsymbol{B})\right)$, giving a total runtime of $\mathcal{O}\left(\frac{n^2}{\ell^2}\lg\kappa(\boldsymbol{B})\left(dn^2+\ell^4\lg\kappa(\boldsymbol{B})\right)\right)$.

Assuming sufficiently many CPU cores available, this leads in theory to an optimal segment size

$$\ell = \Theta(\sqrt[4]{dn^2/\lg\kappa(\boldsymbol{B})}).$$

Already observed in [35, Section 3.5], one should not take ℓ too small, like $\ell = 2$ as was done in [69], because then too many iterations are needed to reach convergence. On the other hand, if ℓ is chosen too large, not only can a single local LLL reduction take a very long time, also there is a larger probability that it does not terminate, or output incorrect values.

Based on experiments and the theoretical optimal value, for dimensions below 1000, we use the following segment size:

$$\ell = 64.$$

The reason to have a single segment size is ease-of-use, while still aiming for near-optimal performance on such dimensions. Moreover, by using this segment size, the algorithm is cache-friendly because each thread uses roughly $16\ell^2$ bytes (64 kiB) of memory, which fits in the cache of many CPUs. Since the segment size is not tied to a power of two, Section 5.4 contains experiments that show the runtime of BLASter as a function of the segment size ℓ.

4 Variants of the BLASter Algorithm

In this section, we provide deep-LLL and BKZ variants of our BLASter algorithm from Section 3, which both achieve a stronger reduction quality than BLASter but require more time.

First, Section 4.1 discusses the deep-LLL variant, which only alters the local LLL reduction. Then, in Section 4.2 we explain how to make a BKZ-like

algorithm, which has an enumeration-based SVP oracle integrated into the segment framework. The BKZ-like algorithm preprocesses the global basis with the deep-LLL variant.

4.1 BLASter with Deep Insertions

The algorithm in Section 3 can easily be adopted to support deep insertions [59] *within segments*. Despite the algorithm not being able to output a deep-LLL-reduced basis, locally running stronger lattice reduction in the leaves of the recursion tree is beneficial for the reduction quality and the convergence speed, as was previously observed by [35,53].

Preferring speed to quality, we still let the algorithm terminate whenever the basis is δ-weakly LLL-reduced (Line 13), even though there may still be a deep insertion possible globally. Hence, the algorithm has the same output guarantees, but on average we expect the basis to be more strongly reduced. Thus, our BLASterDeepLLL algorithm is *almost* the same as Algorithm 2. There are two changes: the algorithm now also expects a depth parameter $1 \leq d \leq \ell$ as input, and Line 7 is changed to

$$\boldsymbol{V}^{(i)} \leftarrow \mathsf{DeepLLLReduce}(\boldsymbol{R}_{[i,j]\times[i,j]}, \delta, d),$$

where DeepLLLReduce is a modification of LLLReduce, that is a floating point LLL algorithm with deep insertions.

Experimentally, we have observed that large depths make the reduction algorithm slower but do not provide significantly stronger bases, so we limit ourselves to a constant depth of $d = 4$.

The implementation of the DeepLLLReduce subroutine is very similar to that of LLLReduce. Now, within a local segment $\boldsymbol{R}' \in \mathbb{R}^{\ell\times\ell}$, the deep-LLL condition, Equation (2), needs to be checked. Suppose that the deep-LLL condition is not satisfied with respect to $1 \leq i < j \leq \ell$, that is,

$$\delta(\boldsymbol{R}'_{ii})^2 > \sum_{k=i}^{j} (\boldsymbol{R}'_{kj})^2.$$

Then, DeepLLLReduce applies the following permutation of basis vectors:

$$\boldsymbol{B}_{[i,j]} \leftarrow [\boldsymbol{b}_j, \boldsymbol{b}_i, \boldsymbol{b}_{i+1}, \ldots, \boldsymbol{b}_{j-1}].$$

In terms of the R-factor, it is updated by iteratively applying Givens rotations on $\boldsymbol{b}_j$, and the preceding basis vector, until $\boldsymbol{b}_j$ is at position i. Givens rotations are particularly easy to implement [30, Chapter 19]. However, when moving to larger segment sizes ℓ, Householder reflections may well be more efficient for updating the local $\boldsymbol{R}$.

Note, that deep-LLL is faster if the basis is preprocessed using LLL, so DeepLLLReduce first calls LLLReduce [19].

Theoretic runtime and output quality. Although the authors of deep-LLL claim it runs in polynomial time when a constant depth is supplied [59], there is no proof known of it [19]. In our iterated local reduction strategy, potentially the amortized cost of deep-LLL may be comparable to the amortized cost of LLL as later iterations may have local blocks that are reduced rather quickly.

If one runs deep-LLL with maximal depth $d = n$, on average it gives a basis with an RHF of 1.012 [46], which is similar to BKZ with block size roughly 20 [23]. Related, BKZ-β with block size $\beta < 25$ mostly finds as an SVP solution the shortest vector to be one of the projected basis vectors, which deep-LLL finds [23, Fig. 14].

4.2 A BKZ-like BLASter Algorithm

Algorithm 3 $\mathsf{BLASterBKZ}(\boldsymbol{B}, \delta, \ell, \ell', \beta, t, P)$

Require: $\delta \in (\frac{1}{4}, 1)$, $\beta \leq \ell'$, and ℓ is even

if $\beta < 40$ **then**
  return $\mathsf{BLASterDeepLLL}\,(\boldsymbol{B}, \delta, \ell, 4)$ ▷ preprocess with depth-4 deep-LLL
$(\boldsymbol{B}, \boldsymbol{U}) \leftarrow \mathsf{BLASterBKZ}\,(\boldsymbol{B}, \delta, \ell, \ell', \beta - P, 1, P)$ ▷ progressive BKZ
$i_0 \leftarrow 0$
while $t > 0$ **do**
  $\boldsymbol{R} \leftarrow \mathsf{QRDecompose}(\boldsymbol{B})$
  $i_0' \leftarrow i_0 \bmod \ell'$
  $\mathcal{I} \leftarrow \{\ (i_0' + k\ell' + 1, \min(n, i_0' + k\ell' + \ell'))\ |\ 0 \leq k < (n - i_0')/\ell'\ \}$
  for all $(i, j) \in \mathcal{I}$ **do in parallel**
    $\boldsymbol{V}^{(i)} \leftarrow \mathsf{BKZReduce}(\boldsymbol{R}_{[i,j]\times[i,j]}, \delta, \beta)$
  for all $(i, j) \in \mathcal{I}$ **do**
    $\begin{bmatrix} \boldsymbol{B}_{[1,d]\times[i,j]} \\ \boldsymbol{U}_{[1,n]\times[i,j]} \end{bmatrix} \leftarrow \begin{bmatrix} \boldsymbol{B}_{[1,d]\times[i,j]} \\ \boldsymbol{U}_{[1,n]\times[i,j]} \end{bmatrix} \cdot \boldsymbol{V}^{(i)}$
  if $i_0 + \beta > n$ **then**
    $(i_0, t) \leftarrow (0, t - 1)$ ▷ one tour is complete
  else
    $i_0 \leftarrow i_0 + (\ell' - \beta + 1)$
  $(\boldsymbol{B}, \boldsymbol{V}) \leftarrow \mathsf{BLASterDeepLLL}\,(\boldsymbol{B}, \delta, \ell, 4)$ ▷ run deep-LLL in between SVP calls
  $\boldsymbol{U} \leftarrow \boldsymbol{U}\boldsymbol{V}$
return $(\boldsymbol{B}, \boldsymbol{U})$

Algorithm 3 lists our BKZ-like algorithm, which makes use of BLASterDeepLLL from Section 4.1. Algorithm 3 provides strong lattice reduction by repeatedly calling an SVP oracle on a projected sublattice of $\boldsymbol{R}$, of a lower dimension β, called the BKZ *block size*. In practice the SVP oracle is faster, when the basis $\boldsymbol{B}$ is preprocessed with LLL, deep-LLL, or even BKZ-β' ($\beta' < \beta$). This SVP oracle can be instantiated either by enumeration [24] or sieving [49,4]. We limit ourselves to a block size of at most $\beta \leq 80$, because sieving is not only

asymptotically more efficient than enumeration (i.e. exponential versus super-exponential), but also in practice due to recent improvements [17,2]. Namely, the latter provides a sieve implementation, `G6K`, that solves SVP-β faster than fplll's pruned enumeration for block sizes $\beta \geq 70$ when using a single thread.

We stress that it may in fact be sufficient *only to run the BLASterDeepLLL with depth* $d = 4$ when a basis of intermediate reduction quality is needed.

BKZ strategy. The BKZ-like algorithm is inspired by *BKZ 2.0* [14], and uses techniques such as early-aborting, sound pruning, preprocessing local blocks and optimizing the enumeration radius. However, no extreme pruning [24] with basis rerandomization is used, so the reduction quality of this algorithm may be less than that of BKZ 2.0 for the same block size [14].

1. We decide beforehand on a fixed number of tours, t, instead of terminating when no shorter vector is found during a whole tour. While drastically reducing the runtime cost, this still achieves a well reduced basis [27].
2. In the algorithm we use pruning parameters to speed up the enumeration subroutine [24], such that the success probability is at least 24%. These pruning parameters can be found using the `Pruning` class in `FPyLLL`, which searches for parameters given a success probability. Because fplll uses an even dimension internally when searching for pruning parameters, we only determine pruning parameters for even block sizes.[2] Odd block sizes β use the pruning parameters of $\beta + 1$ with the last entry removed.
3. Moreover, we choose to preprocess the basis with BLASterDeepLLL from Section 4.1, as it reduces a basis to a slope of -0.043 ± 0.002 in a time comparable to that of Algorithm 2. Such a slope corresponds to an RHF of 1.015 ± 0.001, which is worse than what BKZ-20 gives [13, Table 2.1].
   Then, we use a *progressive BKZ* strategy, with increments of $P = 2$. Block sizes $\beta' < \beta$ perform only one tour. Still, we start progressive BKZ from a block size of at least 40, because enumeration is very fast in dimension 40. Note if BKZ is called with an even block size, then only even block sizes are progressively used, which is convenient by the above comment.
4. The enumeration radius for solving SVP on a dimension-β projected sublattice Λ' of $\boldsymbol{R}$ is:
   $$(1 + \epsilon(\beta))\mathrm{GH}(\beta) \det(\Lambda')^{1/\beta},$$
   where $\epsilon(\beta) = \max(4/\beta, 0.05)$. But we also pick the enumeration radius to be slightly smaller than the norm of the first basis vector of Λ', to ensure a strictly shorter vector is found.

Enumeration. For the enumeration subroutine, we use a well-optimized C++ header file, which is a simplification of the file `lib/enumeration.hpp` from the following repository:

`https://github.com/cr-marcstevens/fplll-extenum`

[2] See `https://github.com/fplll/fplll/blob/a8dedce384689047daba154bd50d6215e35bf03b/fplll/pruner/pruner.h#L253`.

Namely, the enumeration code uses a single thread, and does not find subsolutions. Our enumeration file supports pruning and consists of 151 lines of code.

When given a local R-factor $\boldsymbol{R} \in \mathbb{R}^{\beta\times\beta}$ and enumeration radius r, the subroutine will output a solution vector $\boldsymbol{c} \in \mathbb{Z}^\beta$ such that $\sum_{i=1}^{\beta} \boldsymbol{R}_i \boldsymbol{c}_i$ is the shortest vector in the lattice generated by the columns of $\boldsymbol{R}$, *and* its norm is strictly smaller than r.

Insertion. Suppose we look at the local BKZ block with basis $[\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\beta] = \boldsymbol{R}_{[i,i+\beta-1]\times[i,i+\beta-1]}$, and the enumeration subroutine has found the shortest vector $\boldsymbol{v}' = \sum_{j=1}^{\beta} \boldsymbol{c}_j \boldsymbol{v}_j$ with $\boldsymbol{c}_j \in \mathbb{Z}$. Then necessarily, there is some $\boldsymbol{c}_j \neq 0$, and $\gcd(\boldsymbol{c}) = 1$ because $\boldsymbol{v}'$ is primitive w.r.t. the lattice generated by $[\boldsymbol{v}_1, \ldots, \boldsymbol{v}_\beta]$. To insert $\sum_{j=1}^{\beta} \boldsymbol{c}_j \boldsymbol{b}_{i+j-1}$ into the global basis, one linear relation between the $\beta + 1$ vectors needs to be found. The LLL algorithm can be used for this, but this requires a different memory layout to store the $\beta \times (\beta+1)$ matrix, it is more difficult to implement, and may even be slower in practice.

Instead, we propose a strategy similar to `G6K` [2, Sec. 3.2].[3] We find the largest $j \leq \beta$ such that $\boldsymbol{c}_j \neq 0$ holds, and we require $\boldsymbol{c}_j = \pm 1$ for insertion. If this is not satisfied, we do not perform the insertion. Otherwise, we update $\boldsymbol{R}$ and $\boldsymbol{U}$ as follows:

$$\boldsymbol{R}_{i+j-1} \leftarrow \sum_{k=1}^{\beta} \boldsymbol{c}_k \boldsymbol{R}_{i+k-1}, \qquad \boldsymbol{U}_{i+j-1} \leftarrow \sum_{k=1}^{\beta} \boldsymbol{c}_k \boldsymbol{U}_{i+k-1},$$

and apply repeated Givens rotations to move this new vector from position $i+j-1$ to position i, which keeps $\boldsymbol{R}$ upper triangular. When using a progressive strategy, often the last nonzero $\boldsymbol{c}_j$ is ± 1, so this is a rather mild condition.

Multithreading in BKZ. Similar to the implementations of LLL and deep-LLL, multithreading can also be used within the BKZ-like algorithm for linear algebra, and for running lattice enumeration in parallel segments.

There are two approaches possible for ℓ', the segment size in which SVP calls are made. *Either* $\ell' = \beta$ and one SVP call is done in a segment, *or* $\ell' > \beta$ and multiple SVP calls are done within one segment. If $\ell' > \beta$, the cost of updating the global basis and its transformation matrix $\boldsymbol{U}$ is amortized lower, and much more progress is made during one call (Line 16). However, there are no global deep-LLL calls made in between those calls (only a call to DeepLLLReduce on the segment), and that may impact the reduction quality. Although this may lead one to believe it is best to take $\ell' = \beta$, Algorithm 3 is in practice much quicker with $\ell' > \beta$, and it turns out that the reduction quality is even better. Tacitly, we take $\ell' = 64$ in the experiments with BKZ-60.

When the block size is at least 80 at some point sieving [2] is faster than enumeration [17]. Thus, it is then better to replace the local SVP oracle with a sieving-based one. This would make the codebase of our proof of concept

[3] See `https://github.com/fplll/g6k/blob/88fdac1775585ef2a0269ef29cebf158cd5719d0/g6k/siever.pyx#L1479`

much larger, so we did not implement sieving. It may be more efficient to have a multithreaded sieve in one segment, rather than sieving single-threaded in disjoint segments. Indeed, it seems easier for larger block sizes to use `G6K`, which may call a fast, robust version of BLASter.

5 Experiments

In this section, we show the results of multiple experiments that were performed. There are two regimes that we have experimented with: low dimension and high dimension. In dimensions up to 170, fplll can use `double` precision throughout lattice reduction [65]. Similarly, flatter uses `double` precision when possible.

For the larger dimensions, however, fplll requires multiprecision and becomes much slower. Because the goal of OptLLL and flatter was to perform much better on these instances, it is interesting to compare its performance to BLASter. To carefully keep track of the progress made during lattice reduction, we plot the slope as a function of wall time whenever possible.

First, we explain the used setup in Section 5.1. Second, we benchmark BLASter on low and high dimensional lattices in Section 5.2 and Section 5.2 respectively. Last, in Section 5.4 we experiment with the segment size ℓ to find the optimal runtime, and we investigate which n-dimensional q-ary lattices BLASter is able to reduce.

5.1 Setup

All lattice reduction software was run with the default command line parameters unless otherwise specified, e.g. $\delta = 0.99$, and the segment size was $\ell = 64$.

Lattice generation. Generating a q-ary lattice in *even* dimension n consists of generating a matrix $\boldsymbol{A}$ uniformly at random from the set $(\mathbb{Z}/q\mathbb{Z})^{n/2\times n/2}$, and then constructing the basis $\boldsymbol{B} = \begin{pmatrix} q\,\mathrm{id}_{n/2} & \boldsymbol{A} \\ \boldsymbol{0} & \mathrm{id}_{n/2} \end{pmatrix}$. We generate a set of lattices by invoking the command-line tool 'latticegen' from fplll [68], with a seed ranging from 0 to 9. Specifically, we run the following command:

`latticegen -randseed [seed] q [n] [`$n/2$`] [`q`] q`.

Note that this command line tool, in fact, puts the q-vectors at the end of the basis, so BLASter reverses all the basis vectors when reading such a basis.

Workstation. All experiments were run on a workstation with an Intel(R) Xeon(R) Gold 6248 CPU @ 2.50GHz, which has 40 physical cores (so 80 logical cores with hyperthreading) spread over 2 CPU nodes. All the computations were performed on a single CPU node to have fast memory access on this NUMA system. Moreover, we ran OptLLL and flatter with all 20 threads, while our software used less. In particular, after reading the input lattice, our software automatically uses

$\min(j/2, n/\ell)$ threads, where j is the number of logical threads. This means, in dimension $128, 256, 512$ and 1024, it uses $2, 4, 8$ and 16 threads respectively.

The experiments ran in a virtual environment where NumPy has the common BLAS library OpenBLAS (or specifically: `scipy_openblas64`) as a build dependency.

Slope evolution. For BLASter, BLASterDeepLLL and BLASterBKZ, we are able to track the evolution of the slope in between each iteration, because this can be computed easily from the R-factor. Moreover, we use the environment variable `FLATTER_LOG=[file]` to let flatter report all changes to the basis into a log file. Because we could not find any logging function in fplll and OptLLL, we only report the final slope of the output basis for these.

5.2 Low Dimension

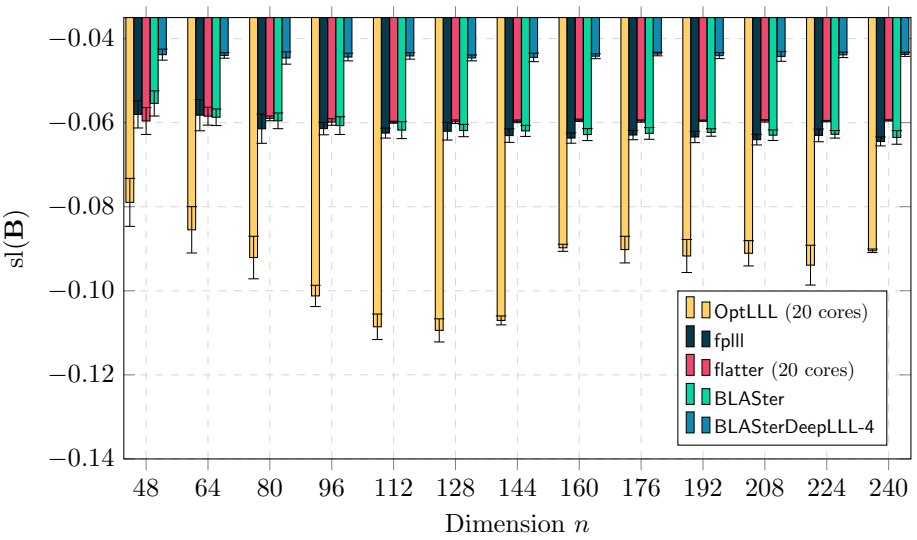

Fig. 1. Slope of various reduction algorithms as a function of dimension. A flatter profile, i.e. a slope closer to zero, corresponds to stronger lattice reduction.

First, we construct various low dimensional q-ary lattices of dimension $n = 48, 64, \ldots$ up to dimension 240 where q is the smallest prime above $2^{n/8}$. With $q > 2^{n/8}$, we expect on average that LLL reduces the whole basis [46], i.e. the first basis vector is reduced to a norm smaller than q, and the last Gram–Schmidt norm is expected to not equal 1. Note while OptLLL and flatter used 20 CPU cores, our algorithm used at most 4 CPU cores.

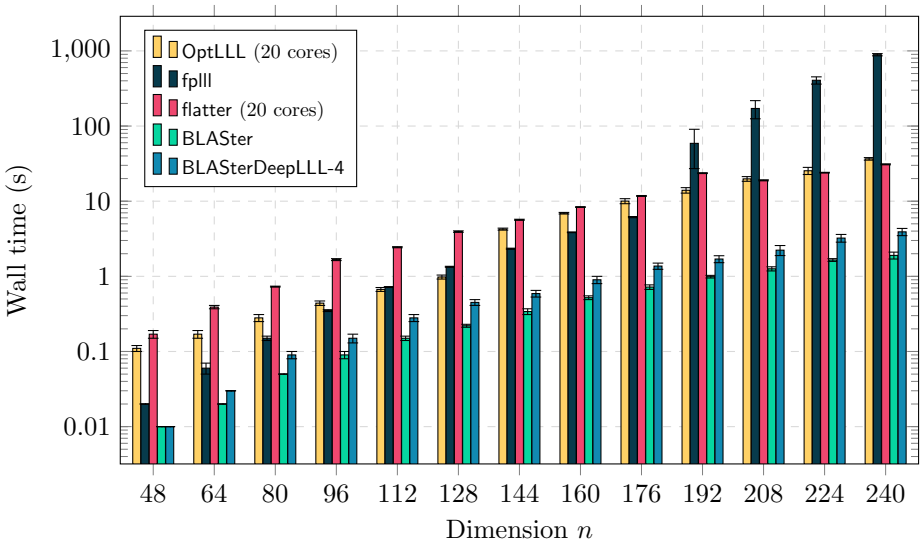

Fig. 2. Used wall time as a function of dimension. The y-axis is logarithmic.

Reduction of this lattice is extensively benchmarked with: BLASter, BLASterDeepLLL with depth $d = 4$, fplll, OptLLL and flatter. The results in terms of the slope and the measured wall time are in Figures 1 and 2 respectively.

Results. While fplll, flatter and our LLL algorithm all achieve a slope of around -0.060, the deep-LLL reduces much stronger to a slope of around -0.043. OptLLL gives a wide range of slopes, all of which are much weaker reduced than average-case LLL.

In dimension 192 and higher, fplll becomes more than a factor 10 slower because it cannot use single-precision doubles and requires multiprecision arithmetic, using Bailey's double-double/quad-double or MPFR data types. Both BLASter and BLASterDeepLLL are more than 10 times faster than flatter. Figure 2 shows that flatter and BLASter handle these larger lattices much better.

Combining Figure 1 with Figure 2, we conclude that doing deep-LLL locally using BLASterDeepLLL only doubles the runtime, whereas the reduction quality improves significantly. Indeed, it is faster than flatter by a factor > 10, and achieves an average slope of -0.043 while flatter achieves a slope of -0.058.

5.3 High Dimension

We ran experiments on lattices in the following dimensions: $n = 128, 256, 512$ and 1024. The value for q is respectively, $631, 829561, 968665207$ and 968665207, which consist of $10, 20, 30$ and 30 bits respectively.[4] Assuming LLL reduction

[4] fplll [68] generates these values for q when running the command 'latticegen q $[n]$ $[n/2]$ $[b]$ p' with $b = 10, 20, 30, 30$ respectively.

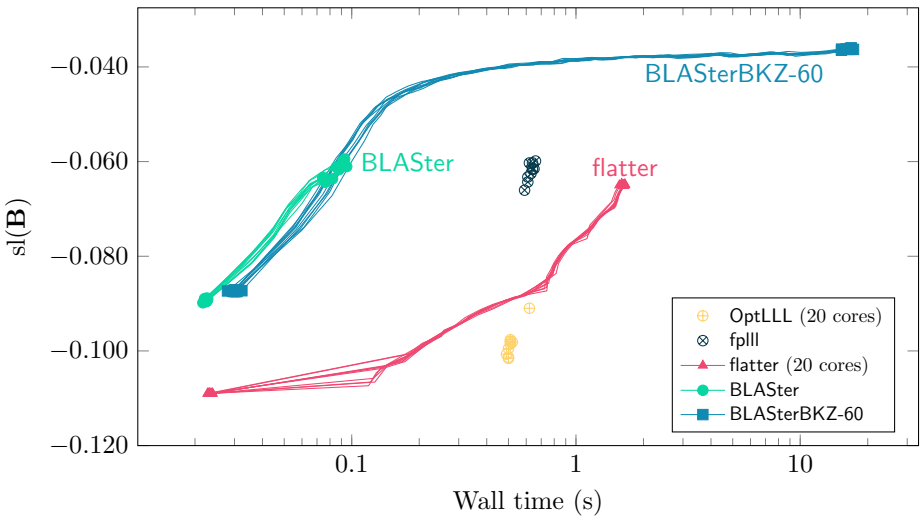

Fig. 3. Evolution of the slope as a function of used wall time for 10 random 631-ary 128-dimensional lattices.

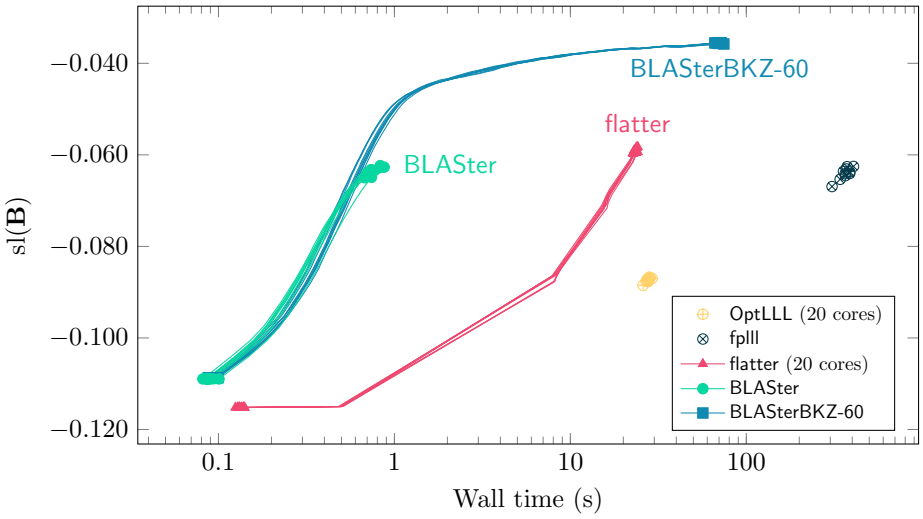

Fig. 4. Evolution of the slope as a function of used wall time for 10 random 829561-ary 256-dimensional lattices.

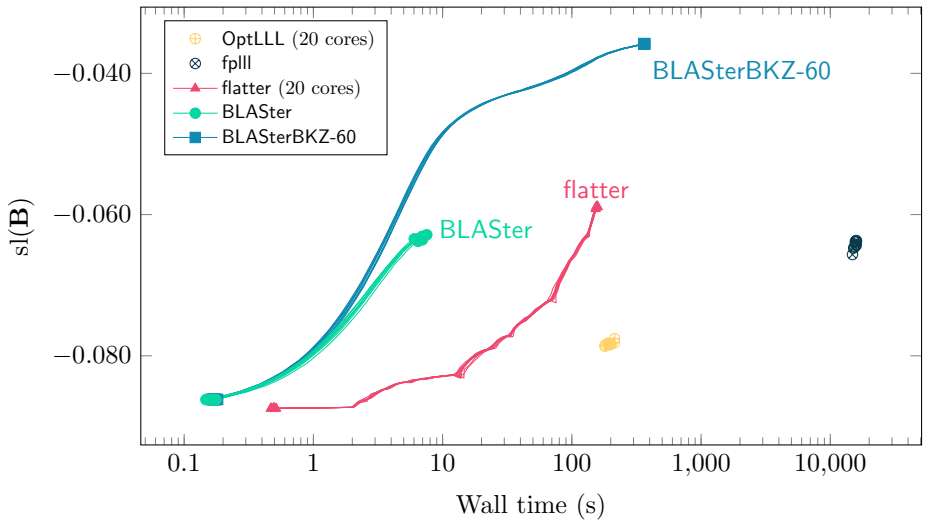

Fig. 5. Evolution of the slope as a function of used wall time for 10 random 968665207-ary 512-dimensional lattices.

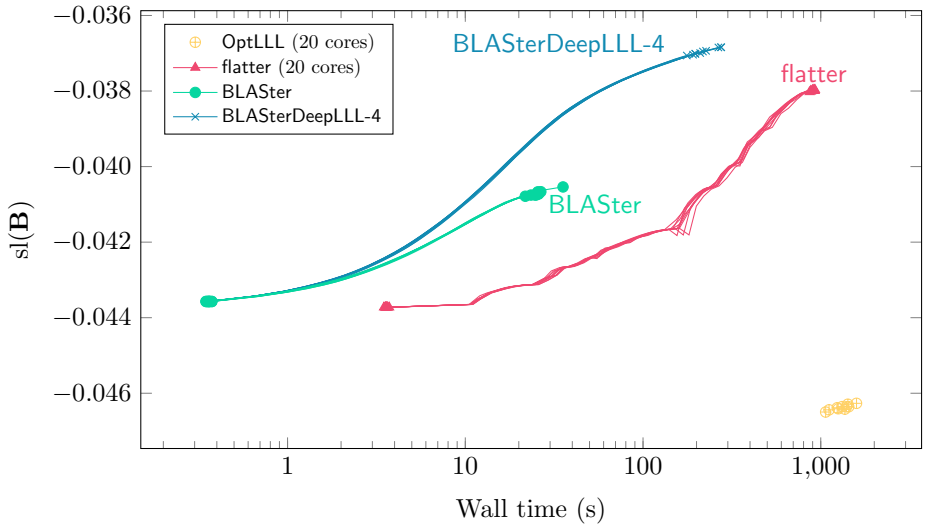

Fig. 6. Evolution of the slope as a function of used wall time for 10 random 968665207-ary 1024-dimensional lattices. flatter was called with argument $\alpha = 0.043$, as with the default parameters, flatter returns without performing any reduction due to the rather small value of q compared to the dimension n.

yields an RHF of 1.0219 [46], the lattices in dimension 128 and 256 will be fully reduced by LLL-like algorithms, whereas a dimension 512 or 1024 basis will keep a couple q-ary vectors at the beginning. We expect BLASterDeepLLL fully reduces the dimension 512 instances, as a slope of -0.058 would already be sufficient.

Moreover, we ran BLASterBKZ with the following arguments: a block size $\beta = 60$, segment sizes $\ell = \ell' = 64$, one tour ($t = 1$) and progressive increments of $P = 2$ in dimensions $128, 256, 512$.

In dimension 1024, the value of q is relatively small, and all algorithms leave some of the initial q-vectors untouched after reduction. Moreover, we did not run fplll to save the extremely long computation. Instead of BLASterBKZ, we ran BLASterDeepLLL as it is still relatively fast.

The Figures 3, 4, 5 and 6 show how the slope evolves as a function of the wall time.

Results. Except for flatter being slower than fplll in dimension 128, we observe that flatter performs better lattice reduction faster than fplll and OptLLL in all high dimensions.

However, flatter is multiple factors slower than BLASter, even though flatter uses more CPU cores than BLASter. In dimensions $128, 256, 512$ the reduction quality is very comparable. In dimension 1024, we forced flatter to reduce to a slope of at least -0.043, which explains why flatter achieves stronger reduction quality as BLASter. Note, however, that BLASterDeepLLL reduces stronger and faster than flatter.

CPU usage. We observed that flatter reaches 100% CPU usage on all assigned cores most of the time, whereas OptLLL runs in fact mostly on a single thread, while all other threads are idle. On the other hand, BLASter is at 100% CPU usage if the full basis is modified. Indeed, if the initial block of the basis contains only q vectors, the local LLL/deep-LLL/BKZ call is immediately done.

In conclusion, it appears that OptLLL does not fully make use of parallelism, whereas flatter seems to be overdoing it.

5.4 Additional Experiments

We conducted two additional experiments using BLASter. First, we experimented with the segment size ℓ used to locally run LLL of this given size. Second, we experimented with the value of q to see whether BLASter, currently not supporting multiprecision, is able to terminate and return a LLL-reduced lattice.

Optimal segment size. Figure 7 depicts the runtime of BLASter, as a function of the segment size ℓ. In dimension $n = 128$ and $n = 256$, the choice for $\ell = 64$ is almost optimal, and there is even a wide range for ℓ that give an optimal runtime. The runtime is much longer for $\ell = 128$ because BLASter uses less CPU cores than for $\ell < 128$, making linear algebra a bottleneck. Forcing a specific number of threads is an easy remedy for this.

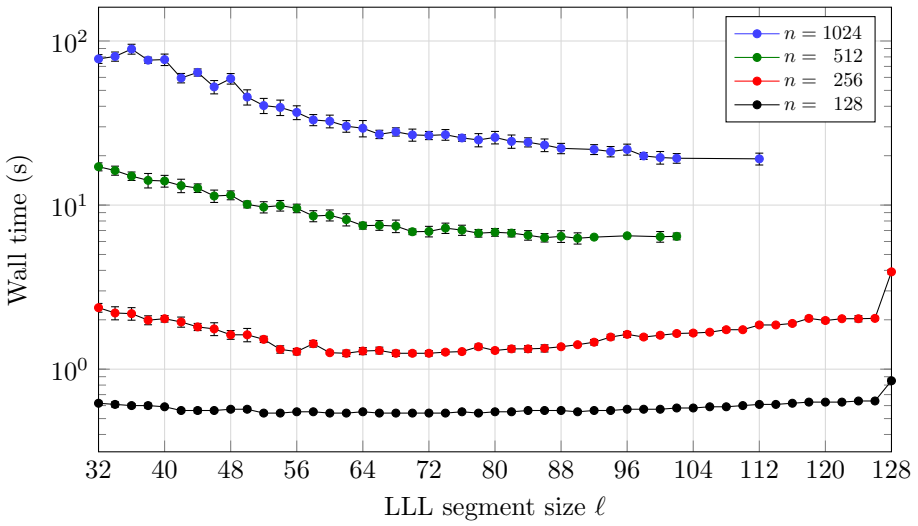

Fig. 7. Wall time of BLASter as a function of the segment size ℓ, averaged over 10 lattices. Each n from a dataset corresponds to the set of 10 lattices of dimension n from Section 5.3.

In dimension $n = 512$ and $n = 1024$, we see that the optimal segment size ℓ is larger than 64, which is in line with the theoretical prediction in Section 3.2. In particular, using $\ell = 96$ instead of $\ell = 64$ reduces the runtime by 13% and 26% respectively in dimension $n = 512$ and $n = 1024$. For segment sizes $\ell > 102$ and $\ell > 112$ in dimension 512 and 1024 respectively, BLASter did not terminate presumably because a local LLL reduction encountered floating point issues, which is more likely for larger segment sizes.

Admissible lattices. The set of q-ary lattices that BLASter is able to reduce, can be found in Figure 8. Experiments ran on 10 lattices that were generated with the following command:

```
latticegen -randseed [seed] q [n] [n/2] [b] p,
```

where seed $\in \{0, 1, \ldots, 9\}$, and $b = \lg(q) \in \{10, 11, \ldots, 64\}$.

In some experiments BLASter did not terminate, mainly because a local LLL reduction never terminated, presumably caused by floating point issues. Therefore, we automatically killed BLASter if it did not terminate within reasonable time, and reported it as a failure.

In addition, BLASter crashed on some instances, which mostly happened during Seysen's reduction. Specifically, most of the errors are thrown on Line 7 of Algorithm 1 because the value of $\boldsymbol{U}'_{12}$ cannot be stored as a signed 64-bit integer. Note that other errors are not reported, such as overflowing a 64-bit integer while updating the basis in Lines 9 and 12 of Algorithm 2.

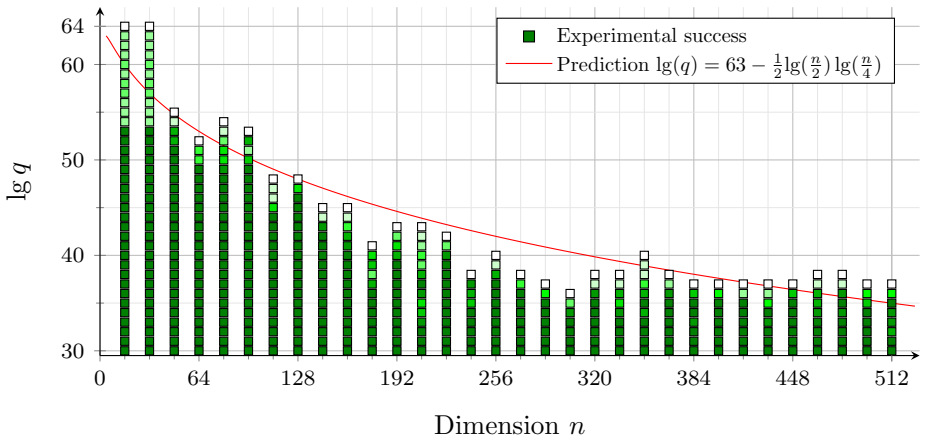

Fig. 8. The set of n-dimensional, q-ary lattices that BLASter is able to reduce. The markers $\{\square, \square, \square, \square, \square, \square, \square, \square, \square, \square, \square\}$ indicate that BLASter is able to reduce $\{0, 1, \ldots, 10\}$ respectively out of 10 randomly sampled n-dimensional q-ary lattices. The prediction is based on an analysis of Seysen's reduction. On average, the first couple basis vectors remain unaltered during LLL reduction of an n-dimensional q-ary lattice basis if $n > 512$ and $\lg q \leq 35$.

The worst-case bound on a Seysen-reduced matrix $\boldsymbol{R}$ with unit diagonal [63, Proposition 5] can be generalized to any upper triangular $\boldsymbol{R}$. Namely, for n a power of two, a Seysen-reduced upper triangular matrix $\boldsymbol{R} \in \mathbb{R}^{n \times n}$ with diagonal bounded by q in absolute value satisfies

$$\|\boldsymbol{R}\|_{\max} \leq q \left(\frac{n}{2} \cdot \frac{n}{4} \cdots 1 \right) = q \cdot 2^{\frac{1}{2}\lg\left(\frac{n}{2}\right)\lg\left(\frac{n}{4}\right)}. \tag{3}$$

If $\|\boldsymbol{R}\|_{\max} > 2^{63}$ holds after Seysen's reduction, then updating $\boldsymbol{B}$ may overflow one of the 64-bit integer entries of $\boldsymbol{B}$ in Line 12 of Algorithm 2. For example, an 64-bit integer entry of $\boldsymbol{B}$ overflows if $\boldsymbol{b}_1$ is a q-vector and the first *column* of $\boldsymbol{R}$ has an entry with an absolute value exceeding 2^{63}. After such an integer overflow in $\boldsymbol{B}$, the subsequent local LLL reduction and the local basis update on Line 9 of Algorithm 2 may not throw errors, and Seysen's reduction will throw an error. This rough calculation yields the prediction in Figure 8, by rewriting it such that the right hand side in Equation (3) is at most 2^{63}.

Being below the prediction curve of Figure 8 is neither a necessary nor a sufficient condition for single-precision BLASter to successfully reduce a q-ary lattice. Still, this prediction roughly approximates the set of q-ary lattices that BLASter can reduce. A more detailed analysis is needed to build a multiprecision variant of BLASter, which may leverage the work of [43] and [12] to determine the required precision for its QR decomposition and Seysen's reduction respectively, and [45] to determine the required precision for storing $\boldsymbol{U}$.

6 Conclusion

In this work, we have implemented an LLL-like algorithm using parallelism, segmentation, Seysen's reduction, and a linear algebra library. This implementation demonstrates that lattice reduction *as is* can still be drastically improved. It is much faster than fplll, OptLLL, and flatter in low dimensions when double precision suffices. Moreover, it can LLL-reduce q-ary lattices in dimension < 1000 with relatively small entries *in a matter of seconds*.

This implementation is just a proof of concept, serving as inspiration and motivation for the development of a robust lattice reduction library that will be significantly faster than fplll. Section 6.1 discusses the speed of a library that uses multiprecision. Section 6.2 shows how this proof of concept is already helpful for a field of experiments that was currently infeasible. Lastly, we mention how low-dimensional slide reduction may benefit from parallelism and Seysen's reduction.

6.1 Support for Multiprecision

The software needs to be extended to support multiprecision, such that all lattices can be reduced. For the robustness, delicate care should be given to the required precision throughout execution, using a strategy that may be similar to fplll and flatter.

Looking at the performance in small dimensions found in Figure 2, we can still expect a huge speed-up over competitors. A heuristic version can be used that increases the precision when encountering numerical stability issues, see § Implementing the L^2 Algorithm [65].

Our main suggestion toward multiprecision while still benefiting from the performance of BLAS would be to completely change the data types. Libraries fplll or flatter construct matrices over GMP [25] and MPFR [20]. Instead, high-precision matrices could be constructed by stacking low-precision matrices as e.g. $\boldsymbol{A} = \sum_{i=0}^{p-1} 2^{32i} \boldsymbol{A}_i$, and applying BLAS on the $\boldsymbol{A}_i$ components.

More detailed implementation considerations are as follows. One could adjust the exponent 32 to allow accumulation of carries before they have to be propagated. Multiprecision multiplication algorithm such as Karatsuba would also pay off much quicker than when applied over integers, because there is a much larger gap between the cost of matrix addition and matrix multiplication. Such a piece of software is a significant task and deserves to be a standalone library, as it could be of use outside of lattice reduction algorithms.

In some sense, the library OptLLL [35] may have gone *too far* in that direction, using an FFT transform of the above representation for even faster multiplication. Among other drawbacks, we note that such representation does not natively allow to perform the rounding steps of size reduction or Seysen's reduction.

6.2 A New Field for Experimentation

Reducing high-dimensional lattices fast opens up a whole new field of experiments. For cryptanalyzing lattice-based cryptographic schemes, a huge effort

has decreased the time needed to run BKZ with high block size. Namely, GPUs can sieve in dimension 100 in a matter of seconds [18] to solve SVP-120 or so. However, the initial lattice reduction would take *over 4 hours* if it is a projected sublattice of a 1000-dimensional lattice!

Currently, experiments on the profile of BKZ-reduced bases with a large block size are limited to lattices of a dimension slightly higher than the block size. However, the HKZ-reduced tail forms a large portion of the whole basis profile, and causes a ripple effect earlier in the basis profile [1]. Thus, integrating (a robust version of) our software with `G6K` [2,18] allows experimenting with high-dimensional high-block size BKZ reductions.

While BLASterBKZ works efficiently for block sizes up to 60, larger block sizes should use a sieving-based SVP oracle, considering the dimensions for free speed-up [17]. There is merit in incorporating `G6K` with our project. As removing the complete dependency on fplll is nontrivial, we leave this integration step as future work.

6.3 Slide Reduction

The slide reduction algorithm [22] could additionally benefit from a segment strategy. One could reduce disjoint primal segments during even iterations in parallel, and disjoint dual segments during odd iterations.

With the strong duality apparent in the slide reduction algorithm, it may be performing well with Seysen's reduction, which controls the size of both primal and dual basis vectors equally. Note however, that sieving is better for large block sizes, so the sieving routine should use parallelism, not the segments.

References

1. Albrecht, M.R., Ducas, L.: Lattice Attacks on NTRU and LWE: A History of Refinements, pp. 15–40. London Mathematical Society Lecture Note Series, Cambridge University Press, Cambridge, United Kingdom (2021). `https://doi.org/10.1017/9781108854207`
2. Albrecht, M.R., Ducas, L., Herold, G., Kirshanova, E., Postlethwaite, E.W., Stevens, M.: The general sieve kernel and new records in lattice reduction. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019, Part II. LNCS, vol. 11477, pp. 717–746. Springer, Cham (May 2019). `https://doi.org/10.1007/978-3-030-17656-3_25`
3. Babai, L.: On Lovász' lattice reduction and the nearest lattice point problem. Combinatorica **6**(1), 1–13 (1986). `https://doi.org/10.1007/BF02579403`
4. Becker, A., Ducas, L., Gama, N., Laarhoven, T.: New directions in nearest neighbor searching with applications to lattice sieving. In: Krauthgamer, R. (ed.) 27th SODA. pp. 10–24. ACM-SIAM (Jan 2016). `https://doi.org/10.1137/1.9781611974331.ch2`
5. Behnel, S., Bradshaw, R., Woods, D., Valo, M., Dalcín, L., et al.: Cython: C-Extensions for Python. `https://cython.org` (2024)
6. Benson, A.R., Ballard, G.: A framework for practical parallel fast matrix multiplication. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles

and Practice of Parallel Programming. p. 42–53. PPoPP '15, Association for Computing Machinery, New York, United States (2015). https://doi.org/10.1145/2688500.2688513
7. Bodrato, M., Zanoni, A.: Integer and polynomial multiplication: towards optimal Toom-Cook matrices. In: Proceedings of the 2007 International Symposium on Symbolic and Algebraic Computation. p. 17–24. ISSAC '07, Association for Computing Machinery, New York, United States (2007). https://doi.org/10.1145/1277548.1277552
8. Bos, J.W., Bronchain, O., Ducas, L., Fehr, S., Huang, Y., Pornin, T., Postlethwaite, E.W., Prest, T., Pulles, L.N., van Woerden, W.: HAWK. Tech. rep., National Institute of Standards and Technology (2023), available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures
9. Brickell, E.F.: Solving low density Knapsacks. In: Chaum, D. (ed.) CRYPTO'83. pp. 25–37. Plenum Press, New York, USA (1983). https://doi.org/10.1007/978-1-4684-4730-9_2
10. Brickell, E.F.: Breaking iterated Knapsacks. In: Blakley, G.R., Chaum, D. (eds.) CRYPTO'84. LNCS, vol. 196, pp. 342–358. Springer, Berlin, Heidelberg (Aug 1984). https://doi.org/10.1007/3-540-39568-7_27
11. Chang, X.W., Stehlé, D., Villard, G.: Perturbation analysis of the QR factor R in the context of LLL lattice basis reduction. Mathematics of Computation **81**(279), 1487–1511 (2012). https://doi.org/10.1090/S0025-5718-2012-02545-2
12. Chang, X.W., Stehlé, D., Villard, G., Wen, J.: Floating-point LLL reduction with a smaller precision. Draft (2021)
13. Chen, Y.: Réduction de réseau et sécurité concrete du chiffrement completement homomorphe. Phd thesis, Université Paris Diderot (Nov 13, 2013), available at https://archive.org/details/PhDChen13
14. Chen, Y., Nguyen, P.Q.: BKZ 2.0: Better lattice security estimates. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 1–20. Springer, Berlin, Heidelberg (Dec 2011). https://doi.org/10.1007/978-3-642-25385-0_1
15. Coppersmith, D.: Small solutions to polynomial equations, and low exponent RSA vulnerabilities. Journal of Cryptology **10**(4), 233–260 (Sep 1997). https://doi.org/10.1007/s001459900030
16. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: Aho, A. (ed.) 19th ACM STOC. pp. 1–6. ACM Press (May 1987). https://doi.org/10.1145/28395.28396
17. Ducas, L.: Shortest vector from lattice sieving: A few dimensions for free. In: Nielsen, J.B., Rijmen, V. (eds.) EUROCRYPT 2018, Part I. LNCS, vol. 10820, pp. 125–145. Springer, Cham (Apr / May 2018). https://doi.org/10.1007/978-3-319-78381-9_5
18. Ducas, L., Stevens, M., van Woerden, W.P.J.: Advanced lattice sieving on GPUs, with tensor cores. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, Part II. LNCS, vol. 12697, pp. 249–279. Springer, Cham (Oct 2021). https://doi.org/10.1007/978-3-030-77886-6_9
19. Fontein, F., Schneider, M., Wagner, U.: PotLLL: a polynomial time version of LLL with deep insertions. DCC **73**(2), 355–368 (2014). https://doi.org/10.1007/s10623-014-9918-8
20. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: A multiple-precision binary floating-point library with correct rounding. ACM Transactions on Mathematical Software (TOMS) **33**(2), 13–es (2007)

21. Frigo, M., Johnson, S.G.: FFTW: An adaptive software architecture for the FFT. In: Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP'98 (Cat. No. 98CH36181). vol. 3, pp. 1381–1384. IEEE (1998)
22. Gama, N., Nguyen, P.Q.: Finding short lattice vectors within Mordell's inequality. In: Ladner, R.E., Dwork, C. (eds.) 40th ACM STOC. pp. 207–216. ACM Press (May 2008). https://doi.org/10.1145/1374376.1374408
23. Gama, N., Nguyen, P.Q.: Predicting lattice reduction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 31–51. Springer, Berlin, Heidelberg (Apr 2008). https://doi.org/10.1007/978-3-540-78967-3_3
24. Gama, N., Nguyen, P.Q., Regev, O.: Lattice enumeration using extreme pruning. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 257–278. Springer, Berlin, Heidelberg (May / Jun 2010). https://doi.org/10.1007/978-3-642-13190-5_13
25. Granlund, T., the GMP development team: GNU MP: The GNU Multiple Precision Arithmetic Library, 5.0.5 edn. (2012), http://gmplib.org/
26. Guennebaud, G., Jacob, B., et al.: Eigen v3 library. http://eigen.tuxfamily.org/ and https://libeigen.gitlab.io/docs/index.html (2010–2023)
27. Hanrot, G., Pujol, X., Stehlé, D.: Analyzing blockwise lattice algorithms using dynamical systems. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 447–464. Springer, Berlin, Heidelberg (Aug 2011). https://doi.org/10.1007/978-3-642-22792-9_25
28. Heckler, C., Thiele, L.: Parallel complexity of lattice basis reduction and a floating-point parallel algorithm. In: Bode, A., Reeve, M., Wolf, G. (eds.) PARLE '93 Parallel Architectures and Languages Europe. vol. 694, pp. 744–747. Springer, Berlin, Heidelberg (1993). https://doi.org/10.1007/3-540-56891-3_74
29. Higham, N.J.: Exploiting fast matrix multiplication within the level 3 BLAS. ACM Transactions on Mathematical Software (TOMS) **16**(4), 352–368 (Dec 1990). https://doi.org/10.1145/98267.98290
30. Higham, N.J.: Accuracy and stability of numerical algorithms. Society for Industrial and Applied Mathematics, United States, second edn. (Aug 1, 2002). https://doi.org/10.1137/1.9780898718027
31. van Hoeij, M.: Factoring polynomials and the knapsack problem. Journal of Number Theory **95**(2), 167–189 (2002). https://doi.org/10.1006/jnth.2001.2763
32. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: Third Algorithmic Number Theory Symposium (ANTS). LNCS, vol. 1423, pp. 267–288. Springer (Jun 1998)
33. Huang, J.: Practical fast matrix multiplication algorithms. Phd thesis, The University of Texas at Austin (Aug 2018), available at http://hdl.handle.net/2152/69013
34. Kaltofen, E.: On the complexity of finding short vectors in integer lattices. In: van Hulzen, J.A. (ed.) European Conference on Computer Algebra. pp. 236–244. EUROCAL '83, Springer, Berlin, Heidelberg, Germany (Mar 28–30, 1983). https://doi.org/10.1007/3-540-12868-9_107
35. Kirchner, P., Espitau, T., Fouque, P.A.: Towards faster polynomial-time lattice reduction. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021, Part II. LNCS, vol. 12826, pp. 760–790. Springer, Cham, Virtual Event (Aug 2021). https://doi.org/10.1007/978-3-030-84245-1_26
36. Koy, H., Schnorr, C.P.: Segment LLL-reduction of lattice bases. In: Silverman, J.H. (ed.) Cryptography and Lattices. pp. 67–80. Springer, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-44670-2_7

37. Koy, H., Schnorr, C.P.: Segment LLL-reduction with floating point orthogonalization. In: Silverman, J.H. (ed.) Cryptography and Lattices. pp. 81–96. Springer, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-44670-2_8
38. Lagarias, J.C., Odlyzko, A.M.: Solving low-density subset sum problems. In: 24th FOCS. pp. 1–10. IEEE Computer Society Press (Nov 1983). https://doi.org/10.1109/SFCS.1983.70
39. Lenstra, A.K., Lenstra, Jr., H.W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische annalen **261**, 515–534 (1982). https://doi.org/10.1007/BF01457454
40. Lyubashevsky, V., Ducas, L., Kiltz, E., Lepoint, T., Schwabe, P., Seiler, G., Stehlé, D., Bai, S.: CRYSTALS-DILITHIUM. Tech. rep., National Institute of Standards and Technology (2022), available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
41. Micciancio, D.: CSE206A: Lattices algorithms and applications (2021), available at https://cseweb.ucsd.edu/classes/fa21/cse206A-a/
42. Minkowski, H.: Geometrie der zahlen. B.G. Teubner, Leipzig, Germany (1896)
43. Morel, I., Stehlé, D., Villard, G.: H-LLL: using householder inside LLL. In: Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation. pp. 271–278. ISSAC '09, Association for Computing Machinery, New York, United States (Jul 28–31, 2009). https://doi.org/10.1145/1576702.1576740
44. Neumaier, A.: Bounding basis reduction properties. DCC **84**(1-2), 237–259 (2017). https://doi.org/10.1007/s10623-016-0273-9
45. Neumaier, A., Stehlé, D.: Faster LLL-type reduction of lattice bases. In: Proceedings of the 2016 ACM International Symposium on Symbolic and Algebraic Computation. pp. 373–380. ISSAC '16, Association for Computing Machinery, New York, United States (Jul 20–22, 2016). https://doi.org/10.1145/2930889.2930917
46. Nguyen, P.Q., Stehlé, D.: LLL on the average. In: Hess, F., Pauli, S., Pohst, M. (eds.) Algorithmic Number Theory. pp. 238–256. ANTS '06, Springer, Berlin, Heidelberg (Jul 23–28, 2006). https://doi.org/10.1007/11792086_18
47. Nguyen, P.Q., Stehlé, D.: An LLL algorithm with quadratic complexity. SIAM Journal on Computing **39**(3), 874–903 (2009). https://doi.org/10.1137/070705702, preliminary version in EuroCrypt 2005.
48. Nguyen, P.Q., Vallée, B. (eds.): The LLL Algorithm - Survey and Applications. ISC, Springer (2010). https://doi.org/10.1007/978-3-642-02295-1
49. Nguyen, P.Q., Vidick, T.: Sieve algorithms for the shortest vector problem are practical. Journal of Mathematical Cryptology **2**(2), 181–207 (2008). https://doi.org/10.1515/JMC.2008.009
50. Novocin, A., Stehlé, D., Villard, G.: An LLL-reduction algorithm with quasi-linear time complexity: extended abstract. In: Fortnow, L., Vadhan, S.P. (eds.) 43rd ACM STOC. pp. 403–412. ACM Press (Jun 2011). https://doi.org/10.1145/1993636.1993691
51. Prest, T., Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z.: FALCON. Tech. rep., National Institute of Standards and Technology (2022), available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
52. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. Journal of the ACM **56**(6), 1–40 (2009). https://doi.org/10.1145/1568318.1568324, preliminary version in STOC 2005.

53. Ryan, K., Heninger, N.: Fast practical lattice reduction through iterated compression. In: Handschuh, H., Lysyanskaya, A. (eds.) CRYPTO 2023, Part III. LNCS, vol. 14083, pp. 3–36. Springer, Cham (Aug 2023). https://doi.org/10.1007/978-3-031-38548-3_1
54. Saruchi, Morel, I., Stehlé, D., Villard, G.: LLL reducing with the most significant bits. In: Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation. pp. 367–374. ISSAC '14, Association for Computing Machinery, New York, United States (Jul 23–25, 2014). https://doi.org/10.1145/2608628.2608645
55. Schnorr, C.P.: A more efficient algorithm for lattice basis reduction. Journal of Algorithms **9**(1), 47–62 (1988). https://doi.org/10.1016/0196-6774(88)90004-1, preliminary version in ICALP 1986.
56. Schnorr, C.P.: Lattice reduction by random sampling and birthday methods. In: Alt, H., Habib, M. (eds.) 20th Annual Symposium on Theoretical Aspects of Computer Science. STACS '03, vol. 2607, pp. 145–156. Springer (Feb 27–Mar 1, 2003). https://doi.org/10.1007/3-540-36494-3_14
57. Schnorr, C.P.: Fast LLL-type lattice reduction. Information and Computation **204**(1), 1–25 (2006). https://doi.org/10.1016/j.ic.2005.04.004
58. Schnorr, C.P.: Progress on LLL and lattice reduction. In: Nguyen and Vallée [48], pp. 145–178. https://doi.org/10.1007/978-3-642-02295-1
59. Schnorr, C.P., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. Mathematical Programming **66**, 181–199 (1994). https://doi.org/10.1007/BF01581144, preliminary version in FCT 1991.
60. Schönhage, A.: Factorization of univariate integer polynomials by diophantine approximation and an improved basis reduction algorithm. In: Paredaens, J. (ed.) Automata, Languages and Programming. pp. 436–447. ICALP '84, Springer, Berlin, Heidelberg, Germany (Jul 1984). https://doi.org/10.1007/3-540-13345-3_40
61. Schönhage, A.: Fast reduction and composition of binary quadratic forms. In: Proceedings of the 1991 International Symposium on Symbolic and Algebraic Computation. pp. 128–133. ISSAC '91, Association for Computing Machinery, New York, United States (1991). https://doi.org/10.1145/120694.120711
62. Schwabe, P., Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Seiler, G., Stehlé, D., Ding, J.: CRYSTALS-KYBER. Tech. rep., National Institute of Standards and Technology (2022), available at https://csrc.nist.gov/Projects/post-quantum-cryptography/selected-algorithms-2022
63. Seysen, M.: Simultaneous reduction of a lattice basis and its reciprocal basis. Combinatorica **13**(3), 363–376 (1993). https://doi.org/10.1007/BF01202355
64. Shoup, V.: NTL: A library for doing number theory. http://www.shoup.net/ntl/index.html (1996–2021)
65. Stehlé, D.: Floating-point LLL: Theoretical and practical aspects. In: Nguyen and Vallée [48], pp. 179–213. https://doi.org/10.1007/978-3-642-02295-1
66. Storjohann, A.: Faster algorithms for integer lattice basis reduction. Technical Report/ETH Zurich, Department of Computer Science **249** (1996)
67. Strassen, V.: Gaussian elimination is not optimal. Numerische Mathematik **13**, 354–356 (1969). https://doi.org/10.1007/BF02165411
68. development team, T.F.: fplll, a lattice reduction library, Version: 5.5.0 (2024), available at https://github.com/fplll/fplll
69. Villard, G.: Parallel lattice basis reduction. In: Papers from the International Symposium on Symbolic and Algebraic Computation. pp. 269–277. ISSAC '92, Association for Computing Machinery, New York, United States (Aug 1992). https://doi.org/10.1145/143242.143327

A Nearest Plane and Size Reduction

In this section, we show an algorithm that on input n targets, computes Babai's NP on all of them in time $\mathcal{O}(n^\omega)$, leading to an amortized time of $\mathcal{O}(n^{\omega-1})$ per target. This directly leads to an algorithm that size reduces a basis using $\mathcal{O}(n^\omega)$ floating point operations. As the time needed to check whether n targets are in $\mathcal{P}(\boldsymbol{B}^*)$ and whether a basis is size-reduced most likely requires a matrix multiplication, these two time complexities are plausibly optimal in a theoretical sense.

This section focuses more on theoretical asymptotic runtimes, but still has potential practical implications. Using this size reduction algorithm in practice, may be faster because it allows one to use fast matrix operations.

Although Seysen's reduction and size reduction have the same runtime complexity, Seysen's reduction was experimentally faster despite the necessary matrix inversion.

A.1 Batch Nearest Plane

Algorithm 4 BatchNearestPlane$\left(\boldsymbol{R} = \begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix}, \quad \boldsymbol{T} = \begin{bmatrix} \boldsymbol{T}_1 \\ \boldsymbol{T}_2 \end{bmatrix}\right)$

if $\boldsymbol{R} \in \mathbb{R}^{1\times 1}$ **then**
  $\boldsymbol{C} \leftarrow \lfloor -\boldsymbol{T}/\boldsymbol{R} \rceil$
  $\boldsymbol{T} \leftarrow \boldsymbol{T} + \boldsymbol{R}\boldsymbol{C}$
  return $\boldsymbol{C}$
else
  $\boldsymbol{C}_2 \leftarrow$ BatchNearestPlane$(\boldsymbol{R}_{22}, \boldsymbol{T}_2)$
  $\boldsymbol{T}_1 \leftarrow \boldsymbol{T}_1 + \boldsymbol{R}_{12}\boldsymbol{C}_2$
  $\boldsymbol{C}_1 \leftarrow$ BatchNearestPlane$(\boldsymbol{R}_{11}, \boldsymbol{T}_1)$
  return $\begin{bmatrix} \boldsymbol{C}_1 \\ \boldsymbol{C}_2 \end{bmatrix}$

Proposition 1. *Algorithm 4, on input an upper triangular matrix $\boldsymbol{R} \in \mathbb{R}^{n\times n}$ and targets $\boldsymbol{T} \in \mathbb{R}^{n\times N}$, outputs a transformation matrix $\boldsymbol{C} \in \mathbb{Z}^{n\times N}$ and modifies $\boldsymbol{T}$ in-place to, say $\boldsymbol{T}'$, such that $\boldsymbol{T}' = \boldsymbol{T} + \boldsymbol{R}\boldsymbol{C}$ and each column of $\boldsymbol{T}'$ is in $\mathcal{P}(\boldsymbol{R}^*)$. If $N \geq n$, its runtime is $\mathcal{O}(Nn^{\omega-1})$ as $n \to \infty$.*

For proving the last line of this proposition, note that multiplication of an $n \times n$ matrix and an $n \times N$ matrix can be done in time $\mathcal{O}(Nn^{\omega-1})$ by splitting the latter matrix in N/n square matrices.

Corollary 1. *Babai's nearest plane runs in amortized time $\mathcal{O}(n^{\omega-1}) = \mathcal{O}(n^{1.38})$ per target, when called with at least n targets.*

The total number of recursive calls in Algorithm 4 is equal to $2n-2$. This can either be proved by induction, or by observing that a rooted, binary tree with n leaves has $n-1$ internal nodes and $2n-2$ edges.

When implementing such an algorithm in practice, the cost of going into a recursive call is noticeable, especially in the deeper layers.

Fortunately, we have written an implemention of Algorithm 4 for the software that is nonrecursive. The idea is to compute the coefficients of $\boldsymbol{C}_{jk}$ $(1 \leq j \leq n, 1 \leq k \leq N)$ by decreasing j, and "lazily" updating $\boldsymbol{T}$. A naïve algorithm will apply the transformation $\boldsymbol{T}_{[j,1:N]} \leftarrow \boldsymbol{T}_{[j,1:N]} + \boldsymbol{R}_j[\boldsymbol{C}_{j1} \dots \boldsymbol{C}_{jN}]$ directly, but that will not give an $\mathcal{O}(Nn^{\omega-1})$ complexity. Instead, note that each of the $n-1$ recursive calls on submatrices of the form $\boldsymbol{R}_{[i,k)\times[i,k)}$ $(1 \leq i < k \leq n)$, recursively call themselves on the submatrices $\boldsymbol{R}_{[i,j)\times[i,j)}$ and $\boldsymbol{R}_{[j,k)\times[j,k)}$, where j is halfway i and k. Conversely, for all $1 < j \leq n$, there is a unique pair (i,k) such that the above holds, and by determining these (i,k) for all j, we can now lazily update $\boldsymbol{T}$ as follows.

$$\boldsymbol{T}_{[i,j)\times[1,N]} \leftarrow \boldsymbol{T}_{[i,j)\times[1,N]} + \boldsymbol{R}_{[i,j)\times[j,k)}\boldsymbol{C}_{[j,k)\times[1,N]}.$$

A.2 Size Reduction

Algorithm 5 SizeReduce($\boldsymbol{R}$)

1: **if** $\boldsymbol{R} \in \mathbb{R}^{1\times 1}$ **then**
2: **return** $\begin{bmatrix} 1 \end{bmatrix}$
3: Parse $\begin{bmatrix} \boldsymbol{R}_{11} & \boldsymbol{R}_{12} \\ \boldsymbol{0} & \boldsymbol{R}_{22} \end{bmatrix} \leftarrow \boldsymbol{R}$ ▷ Blocks have half the size
4: $\boldsymbol{U}_{11} \leftarrow$ SizeReduce($\boldsymbol{R}_{11}$)
5: $\boldsymbol{U}_{22} \leftarrow$ SizeReduce($\boldsymbol{R}_{22}$)
6: $\boldsymbol{R}_{12} \leftarrow \boldsymbol{R}_{12}\boldsymbol{U}_{22}$
7: $\boldsymbol{U}_{12} \leftarrow$ BatchNearestPlane($\boldsymbol{R}_{11}, \boldsymbol{R}_{12}$)
8: **return** $\begin{bmatrix} \boldsymbol{U}_{11} & \boldsymbol{U}_{12} \\ \boldsymbol{0} & \boldsymbol{U}_{22} \end{bmatrix}$

Now, using Babai's NP algorithm, we can build a recursive block variant of size reduction, which is listed in Algorithm 5. We want to remark that it is surprisingly similar to Seysen's reduction, Algorithm 1. Indeed, the first 5 lines are the same, but the more preciseful size reduction of $\boldsymbol{R}_{12}$ is done differently. Instead of adding $\boldsymbol{R}_{11}\left\lfloor -\boldsymbol{R}_{11}^{-1}\boldsymbol{R}_{12} \right\rceil$ to $\boldsymbol{R}_{12}$, NP is called.

Proposition 2. *There exists an $\mathcal{O}(n^{\omega})$-time algorithm that, on input an* upper triangular *basis $\boldsymbol{R} \in \mathbb{R}^{n\times n}$, outputs an* upper triangular *unimodular matrix $\boldsymbol{U} \in \mathbb{Z}^{n\times n}$ and modifies $\boldsymbol{R}$ in-place to, say $\boldsymbol{S}$, such that $\boldsymbol{S} = \boldsymbol{R}\boldsymbol{U}$ and $\boldsymbol{S}$ is size-reduced.*

Proof. The proof is easily proven by induction. To prove $\boldsymbol{S}$ is size-reduced, note that the columns of $\boldsymbol{S}_{12}$ lie in $\mathcal{P}(\boldsymbol{S}_{11}^{*})$.

We have implemented a version Algorithm 5 that does not use recursion.

Practicality. Similar to [35], the runtime complexity of these algorithms cannot be reached in practice, because theoretically fast matrix multiplication algorithms are not fast in practice. These algorithms require more workspace, are generally not cache-friendly [33], and may also be numerically less stable.

Still, using practically fast matrix multiplication from the BLAS library will give good performance. Experimentally, we did not see large numerical issues when using the size reduction of Algorithm 5 instead of Seysen's reduction, when reducing the lattices in Section 5.

B Block Cholesky Decomposition

In this section, we provide an optimization to [35, Algorithm 6] that instead of 4 matrix multiplications (and more recursively) only uses 2 matrix multiplications.

Note [35, Algorithm 6] contains a typo: Line 5 should read $\boldsymbol{A}' = \boldsymbol{R}'_A(\boldsymbol{R}'_A)^\mathsf{T}$, making $\boldsymbol{A}' = \boldsymbol{A}^{-1}$.

Algorithm 6 is based on [35, Algorithm 6]. Similarly, we do not provide details of the numerical precision required for correctness.

Algorithm 6 $\mathsf{BlockCholesky}(\boldsymbol{G})$

1: **if** $\dim(\boldsymbol{G}) = 1$ **then**
2: **return** $\sqrt{\boldsymbol{G}}$
3: Parse $\begin{pmatrix} \boldsymbol{A} & \boldsymbol{B} \\ \boldsymbol{B}^\mathsf{T} & \boldsymbol{C} \end{pmatrix} \leftarrow \boldsymbol{G}$ ▷ Blocks have half the size
4: $\boldsymbol{R}_A \leftarrow \mathsf{BlockCholesky}(\boldsymbol{A})$
5: $\boldsymbol{S} \leftarrow (\boldsymbol{R}_A^\mathsf{T})^{-1}\boldsymbol{B}$
6: $\boldsymbol{R}_C \leftarrow \mathsf{BlockCholesky}(\boldsymbol{C} - \boldsymbol{S}^\mathsf{T}\boldsymbol{S})$
7: **return** $\begin{pmatrix} \boldsymbol{R}_A & \boldsymbol{S} \\ \boldsymbol{0} & \boldsymbol{R}_C \end{pmatrix}$.

Proposition 3. *Algorithm 6 on input a positive-definite symmetric $\boldsymbol{G} \in \mathbb{R}^{n\times n}$, outputs an upper triangular $\boldsymbol{R}$ such that $\boldsymbol{G} = \boldsymbol{R}^\mathsf{T}\boldsymbol{R}$ in time $\mathcal{O}(n^\omega)$.*

Proof. Let us consider the nontrivial case $n > 1$. By induction, $\boldsymbol{R}_A$ satisfies $\boldsymbol{R}_A^\mathsf{T}\boldsymbol{R}_A = \boldsymbol{A}$. Moreover, we have $\boldsymbol{R}_A^\mathsf{T}\boldsymbol{S} = \boldsymbol{R}_A^\mathsf{T}(\boldsymbol{R}_A^\mathsf{T})^{-1}\boldsymbol{B} = \boldsymbol{B}$, and

$$\boldsymbol{S}^\mathsf{T}\boldsymbol{S} + \boldsymbol{R}_C^\mathsf{T}\boldsymbol{R}_C = \boldsymbol{S}^\mathsf{T}\boldsymbol{S} + \left(\boldsymbol{C} - \boldsymbol{S}^\mathsf{T}\boldsymbol{S}\right) = \boldsymbol{C},$$

so the return value, say $\boldsymbol{R}$, satisfies $\boldsymbol{R}^\mathsf{T}\boldsymbol{R} = \boldsymbol{G}$.

The runtime is clearly bounded by the matrix multiplication on matrices of size $n/2$, and inverting the upper triangular $\boldsymbol{R}_A$, which takes time $\mathcal{O}(n^\omega)$ [35, Lemma 1].