A Greedy Global Framework for LLL

Sanjay Bhattacherjee[1], Julio Hernandez-Castro[1], Jack Moyler[1*]

[1*]Institute of Cyber Security for Society and School of Computing, University of Kent, Canterbury, CT2 7NP, Kent, United Kingdom.

*Corresponding author(s). E-mail(s): jdm58@kent.ac.uk;
Contributing authors: s.bhattacherjee@kent.ac.uk;
j.c.hernandez-castro@kent.ac.uk;

Abstract

LLL-style lattice reduction algorithms iteratively employ size reduction and reordering on ordered basis vectors to find progressively shorter, more orthogonal vectors. These algorithms work with a designated measure of basis quality and perform reordering by inserting a vector in an earlier position depending on the basis quality before and after reordering. **DeepLLL** was introduced alongside the **BKZ** reduction algorithm, however the latter has emerged as the state-of-the-art and has therefore received greater attention. We first show that **LLL**-style algorithms iteratively improve a basis quality measure; specifically that **DeepLLL** improves a sublattice measure based on the generalised Lovász condition. We then introduce a new generic framework for lattice reduction algorithms, working with some quality measure $\boldsymbol{X}$. We instantiate our framework with two quality measures - basis potential (**Pot**) and squared sum (**SS**) - both of which have corresponding **DeepLLL** algorithms. We prove polynomial runtimes for our $\boldsymbol{X}$-**GGLLL** algorithms and guarantee their output quality. We run two types of experiments (implementations provided publicly) to compare performances of **LLL**, $\boldsymbol{X}$-**DeepLLL**, $\boldsymbol{X}$-**GGLLL**; with multi-precision arithmetic using overestimated floating point precision for *standalone* comparison with no preprocessing, and with standard datatypes using **LLL**-*preprocessed* inputs. In preprocessed comparison, we also compare with **BKZ**. In standalone comparison, our **GGLLL** algorithms produce better quality bases whilst being much faster than the corresponding **DeepLLL** versions. The runtime of **SS-GGLLL** is only second to **LLL** in our standalone comparison. **SS-GGLLL** is significantly faster than the FPLLL implementation of **BKZ-12** at all dimensions and outputs better quality bases dimensions **100** onward.

Keywords: Lattice reduction, LLL, DeepLLL, greedy global framework, potential, squared sum.

1 Introduction

A Euclidean lattice (or just a lattice) $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^m$. It can be represented by a basis matrix $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n) \in \mathbb{R}^{m \times n}$ made of linearly independent column vectors $\mathbf{b}_i \in \mathbb{R}^m$ such that $\mathcal{L} = \{\mathbf{Bx} : \mathbf{x} \in \mathbb{Z}^n\}$. There are infinitely many bases for any lattice with $n \geq 2$ and there are ways to transform a basis into another for the same lattice. The quality of a given lattice basis is determined by the length of the vectors and how close to orthogonal they are to each other. Bases with shorter and more orthogonal vectors are considered to be of better quality. Given a lattice specified by a basis, finding a good quality basis and short vectors therein is of major importance. The process of transforming a given basis into one of better quality is generally called lattice reduction.

In 1982, Lenstra, Lenstra, and Lovász [1] presented the first lattice reduction algorithm that came to be called LLL after its inventors. LLL uses the Gram-Schmidt orthogonalisation (GSO) $\mathbf{B}^* = (\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*)$ of the basis $\mathbf{B}$. The GSO process assumes an inherent ordering of the vectors and LLL works with the same order. Starting from index $k = 2$ of the ordered basis, LLL traverses up and down the order in a loop by incrementing or decrementing the index k by 1 in each iteration. There are two kinds of operations – size reductions and swaps – that are executed within the loop until the entire basis is of sufficiently good quality. The quality of the basis is determined by the optimisation criterion called the Lovász condition (LC) on all pairs of consecutive vectors $\mathbf{b}_{k-1}, \mathbf{b}_k \in \mathbf{B}$. This condition is given by $\left\|\mathbf{b}_k^* + \mu_{k,k-1}\mathbf{b}_{k-1}^*\right\|^2 \geq \delta \left\|\mathbf{b}_{k-1}^*\right\|^2$, where the $\mu_{i,j}$'s are the GSO coefficients and $1/4 < \delta \leq 1$ is a parameter which determines the quality of each reduction. The quality improves as δ increases. After LLL terminates, vector $\mathbf{b}_i$ in the output basis is an exponential approximation of the i^{th} shortest linearly independent vector in the lattice. In [1], LLL was shown to run in polynomial time using an argument surrounding a quantity known as the *potential* of the basis – a measure of basis quality that we will describe soon. LLL has many applications including in cryptology [2], algorithmic number theory [3], factoring polynomials [4], Diophantine approximation [5] etc.

Schnorr and Euchner introduced a variant of the LLL algorithm called LLL with deep insertions, or DeepLLL [6]. The key algorithmic novelty was in the reordering of the vectors. They introduced the notion of deep insertions whereby instead of just swapping vector $\mathbf{b}_k$ with the immediate previous vector $\mathbf{b}_{k-1}$, it could be inserted before any one of the previous vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{k-1}$. This essentially meant that the index k could be decremented to any value between $\{2, \ldots, k-1\}$. They also extended the LC-constraint from consecutive pairs $(\mathbf{b}_{k-1}, \mathbf{b}_k)$ to all pairs $(\mathbf{b}_i, \mathbf{b}_k)$ for $i < k$ in the ordering[1]. This introduced more constraints on the output basis and as a result, the quality of the output basis is provably better [7] than in LLL. In other words, the i^{th} vector of the output basis is a better approximation of the i^{th} shortest linearly independent vector of the lattice, as compared to the LLL output [7, Theorem 1]. However, DeepLLL requires additional size reduction steps and bookkeeping that makes it significantly more time-consuming than LLL.

[1] A pair $(\mathbf{b}_i, \mathbf{b}_k)$ in a basis can simply be identified by the pair of indices (i, k).

In the same paper [6], the authors introduced an algorithm which performs the block Korkin-Zolotarev (BKZ) reduction. Since this paper, BKZ has become state-of-the-art in lattice reduction and has thus been more researched than DeepLLL-style algorithms. BKZ takes as input a parameter β denoting the *block size*. It iteratively reduces consecutive projected blocks of size β by calling a shortest vector problem (SVP) oracle on these blocks and inserting the resultant vector into the basis. As β increases, the algorithm outputs bases of better quality, however the runtime also increases. Improvements were made to BKZ in [8] by incorporating a pruning technique on the enumeration SVP subroutine which decreases the runtime without any decrease in basis quality on output. Furthermore, the authors introduced preprocessing of the local bases and reducing the enumeration radius; both to reduce the runtime of the enumeration subroutine.

Since Schnorr and Euchner introduced DeepLLL, there have been two new deep-insertion based algorithms - Pot-LLL [9] and SS-LLL [7]. These algorithms replace the extended Lovász condition of DeepLLL with a check on the improvement of a basis quality. They use the quality measures *potential* $\mathrm{Pot}(\mathbf{B}) = \prod_{i=1}^{n} \|\mathbf{b}_i^*\|^{2(n-i+1)}$ and *squared sum* $\mathrm{SS}(\mathbf{B}) = \sum_{i=1}^{n} \|\mathbf{b}_i^*\|^2$ respectively, computed directly from the Gram-Schmidt orthogonalised basis $\mathbf{B}^*$. To stress that they are essentially variants of DeepLLL, we call them Pot-DeepLLL and SS-DeepLLL respectively. They are both polynomial-time algorithms that provide efficiency versus basis quality trade-offs in between LLL and DeepLLL. They typically find shorter vectors than LLL but not as short as DeepLLL. They are slower than LLL, but faster than DeepLLL.

In every iteration of DeepLLL and its variants Pot-DeepLLL and SS-DeepLLL, the algorithms only work with the sublattice $\mathcal{L}_k$ generated by a subset $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ of $\mathbf{B}$. We note that each of these algorithms attempts to iteratively improve some basis quality measure. The use of $\mathrm{Pot}(\cdot)$ and $\mathrm{SS}(\cdot)$ as measures of quality has been quite clear in the proofs of basis quality and runtime complexity of these algorithms. However, DeepLLL has not been interpreted as or represented in a form where it is improving an explicit quality measure in every iteration, to the best of our knowledge. We do this exercise of interpreting the (generalised) Lovász condition as a reordering constraint used to improve the length $\|\mathbf{b}_i^*\|$ of the i^{th} GSO vector of the basis, which is a localised measure of the quality of the basis. In contrast, $\mathrm{Pot}(\cdot)$ and $\mathrm{SS}(\cdot)$ are global measures on the entire basis. We thus have a generalised understanding of all three algorithms based on deep insertions looking to improve quality measures of a basis.

The above generalisation leads us to our new generic framework of algorithms. We ask the following question – *if the general principle of* LLL*-style algorithms is to iteratively improve the basis quality, is there a greedy approach to improve it as much as possible at any point?*

Previous LLL-style algorithms [1, 6, 9, 7] maintain an index k of the vector to be inserted at a previous position $i \in \{1, \ldots, k-1\}$ in the basis ordering, to improve the basis quality. In LLL [1], $i = k-1$ is a fixed previous position for a certain k, while in algorithms using deep insertions [6, 9, 7], the choices for the deep insertion position i are restricted within the sublattice $\mathcal{L}_k$ in an iteration. A (deep) insertion of $\mathbf{b}_k$ at a position i can be substituted by a (deep) insertion of $\mathbf{b}_{k'}$ at position i' for $k' < k$ or $k' > k$, and $i' \in \{1, \ldots, k'-1\}$, such that the basis quality improvement is better.

In fact, there is a pair (i', k') for which the improvement in quality is the maximum possible at that point. This observation is the basis of our greedy choice.

In this work, we move away from the technique of maintaining an index k and working with a sublattice $\mathcal{L}_k$. We propose a new generic framework for lattice reduction through an algorithm X-GGLLL. Our algorithm works with a general quality measure $X(\mathbf{B})$ of the basis. To iteratively improve the quality of the basis, *we make a dynamic greedy choice of a pair of indices (i, k), $1 \leq i < k \leq n$ globally over the entire basis such that the deep insertion of $\mathbf{b}_k$ at position i minimises the basis quality measure* $X(\cdot)$. Such deep insertions are carried out as long as the measure of the reordered basis decreases by *at least* a fraction $(1 - \delta)$ of its previous value. When the algorithm terminates, the output basis is guaranteed to have a measure that can not be reduced appreciably (by a fraction $(1 - \delta)$) any further through deep insertions. For a measure X, we call such a basis δ-X-DeepLLL reduced. *By choosing the maximum change in $X(\cdot)$ possible at each iteration, our greedy algorithm reaches such a state in a small (if not the smallest) number of iterations.* When the measure has a positive lower bound, the algorithm is guaranteed to terminate.

The choice of the measure $X(\cdot)$ is a key determining factor in the framework of algorithms we propose. We instantiate our generalised algorithm X-GGLLL with the measures $\mathrm{Pot}(\cdot)$ and $\mathrm{SS}(\cdot)$ in place of $X(\cdot)$ to get the Pot-GGLLL and SS-GGLLL algorithms respectively. We prove that X-GGLLL outputs a δ-X-DeepLLL reduced basis and provide theoretical bounds on the runtime of X-GGLLL. We prove the concrete polynomial runtime complexities for both Pot-GGLLL and SS-GGLLL and show that they are the same as their X-DeepLLL counterparts.

We conduct extensive experiments using floating-point implementations. We assess the performance of our algorithms in two ways – (1) by running them on bases that have been *preprocessed* with 0.99-LLL (i.e. LLL with $\delta = 0.99$), and (2) as *standalone* algorithms running on bases that have not been preprocessed in any way. For the standalone comparison, we use implementations with multi-precision data types that can hold the large integer elements from the input bases, and the subsequent floating point computations with them. We use overestimated floating-point precisions for each dimension, to ensure correctness. As the dimension increases, the overestimated precision is increased and consequently the algorithms get slower. It may be possible to improve the runtimes of the X-DeepLLL and X-GGLLL algorithms using techniques from L^2 [10]; however, these have not been explored in this work. The preprocessed comparison is using implementations with standard data types. The output quality of the algorithms does not vary between our experimental setups. So the two setups are essentially to compare the runtime performances of the algorithms in the two cases.

We report our results in three parts. First, we perform a comparison of the output quality of LLL [1], Pot-DeepLLL [9], SS-DeepLLL [7], BKZ [6] with blocksizes $8, 10, 12$ and 20, and our greedy global algorithms Pot-GGLLL and SS-GGLLL in dimensions $40 - 210$. We then provide the corresponding runtime comparisons. We first compare the standalone runtimes of the LLL-style algorithms (LLL, X-DeepLLL and X-GGLLL) in dimensions 40 to 150. Finally we compare the preprocessed runtimes of the X-DeepLLL and X-GGLLL algorithms and BKZ with blocksize 8, 10, 12 and 20 in dimensions 40 to 210.

The following are the key findings from our experiments.

- X-DeepLLL versus X-GGLLL: standalone and preprocessed comparison.
  - The output quality of X-DeepLLL and X-GGLLL algorithms does not vary between the standalone and the preprocessed experiments. So we report them once.
  - The output quality of X-GGLLL is better than the corresponding X-DeepLLL algorithm, as can be seen from Tables 2 and 3 and Figures 1, 2, and 3. For example, at dimension 210, SS-GGLLL outputs shortest vectors that are on average 11.6% shorter than those by SS-DeepLLL.
  - In standalone comparison with overestimated precision, the X-GGLLL algorithms are much faster than the corresponding X-DeepLLL algorithms. Furthermore, as the dimension grows, *the X-GGLLL algorithms become even better in comparison.* At dimension 150, SS-GGLLL is around 2.3 times faster than SS-DeepLLL and Pot-GGLLL is about 1.4 times faster than Pot-DeepLLL. We provide intuitive experimental justification by showing that X-GGLLL *requires significantly fewer deep insertions and overall fewer size reductions than* X-DeepLLL.
  - In preprocessed comparisons with standard data types, the X-GGLLL algorithms are slower than the corresponding X-DeepLLL algorithms. We again provide intuitive experimental justification by showing that *even though* X-GGLLL *requires significantly fewer deep insertions than* X-DeepLLL*, it is unable to compensate for the increased number of size reductions.*
- X-GGLLL versus BKZ: preprocessed comparison only.
  - SS-GGLLL has better output quality than BKZ-8 throughout, BKZ-10 around and beyond dimension 60, and BKZ-12 around and after dimension 100. Our standard data type implementation of SS-GGLLL is around 6 times faster than the corresponding FPLLL BKZ-12 implementation. In other words, SS-GGLLL is better than BKZ-12 both in runtime as well as output quality.
  - In terms of runtime, whilst it was reported in [9] that Pot-DeepLLL has a runtime comparable to BKZ-5, there was no claim in [7] regarding the runtime performance of SS-DeepLLL compared with BKZ. Our experiments show the surprising result that the X-DeepLLL as well as the X-GGLLL algorithms are faster than BKZ with $\beta \geq 8$ across all tested dimensions on the preprocessed bases. We conjecture that the incorporation of the GSO update techniques from [11] is perhaps the main reason behind this excellent runtime performance of X-DeepLLL and subsequently X-GGLLL.

Our implementations, the input bases we have used in our experiments and the outputs of said experiments are available at [12]. Other than our own algorithms, this repository has the only publicly available implementation of Pot-DeepLLL and SS-DeepLLL with the incorporation of the GSO update techniques from [11], to the best of our knowledge.

The outline of the paper is as follows. Section 2 details the relevant notation and gives an overview of lattices. Section 3 provides a description of LLL and generalises DeepLLL for any measure. Section 4 proposes the greedy global framework as a

novel way of reducing lattice bases. Sections 5 and 6 provide theoretical analysis and experimental results, respectively.

Related Works.

Yamaguchi and Yasuda in [11] described an efficient algorithm for updating the GSO information in DeepLLL. Since the update of the GSO information is dominant in such an algorithm, this work is of great importance to our framework. In [13], it was proved that in LLL the value of the squared sum SS($\mathbf{B}$) decreases with every swap. The complexity of LLL [1] and Pot-DeepLLL [9, Proposition 1] for an input basis $\mathbf{B}$ is bounded by the size of Pot($\mathbf{B}$). The complexity of SS-DeepLLL is bounded by the size of SS($\mathbf{B}$) [7]. Fukase and Kashiwabara [14] showed that a basis with a smaller squared-sum allows more short lattice vectors to be sampled using Schnorr's random sampling. This method was used in [13] to sample short vectors.

The original LLL algorithm [1] was known to run in polynomial time for the reduction parameter $\delta < 1$. For $\delta = 1$, it is known to be polynomial time, but only for fixed dimensions [15]. Although DeepLLL [6] is not known to run in polynomial time, its variants Pot-DeepLLL [9] and SS-DeepLLL [7] are both polynomial time algorithms.

The potential was analysed in [16], where the author examined LLL reduction based on maximally reducing the basis potential for a given lattice. Whilst LLL continues until the potential can not be further reduced by a factor of δ, this does not mean that LLL reduces the potential maximally. Instead, the author introduces a new notion of basis reduction whose aim is to find the basis $\mathbf{B}$ such that the potential of $\mathbf{B}$ is smaller than the potential of all other bases $\mathbf{B}'$ for the same lattice. In [17], the authors pointed out that Pot($\cdot$) does not capture the typical unbalancedness demonstrated by the GSO norms. They introduced a new potential function based on the sublattice $\mathcal{L}_k$ generalising the one depending on the entire basis and demonstrating their usefulness.

An important direction in improving the efficiency of LLL has been considering the implementation details of the algorithms, floating-point arithmetic considerations and their consequent optimisations. In [6], a practical LLL algorithm using floating-point arithmetic was described, which has been extended by Nguyen and Stehlé [10] in their very efficient L^2 algorithm. L^2 is a significant improvement in LLL reduction, where careful management of the precision required for the floating-point computation of the Gram-Schmidt orthogonalisation (GSO) information results in a more efficient algorithm with no reduction in output quality. This is an important research direction in lattice basis reduction. Another variant of LLL was introduced in [18], where the costly GSO computations are approximated by Householder transformations which are performed using floating-point arithmetic. In [19], a perturbation analysis has been performed on the $\mathbf{QR}$ factor $\mathbf{R}$ of LLL-reduced bases under columnwise perturbation. The results obtained may be applied to the floating-point implementations of LLL-type algorithms. LLL has also been adapted to obtain a small speed-up in the Information-Set Decoding algorithm for binary codes in [20]. Lenstra introduced the idea of a flag in lattice reduction in [21], where a flag is defined to carry a little less information than is provided by a lattice basis. The flag is then reduced within the LLL algorithm by performing successive steps which replace flags with neighbouring ones for reduction.

In [22], the authors used parallelisation and recursion to improve the efficiency of LLL by decreasing the precision required for reduction as the basis is reduced. An improved variant of LLL using recursion is described in [23], where a notion of reduction based on the drop of the profile of a lattice is introduced. The authors also introduce a novel method for precision management in their algorithm and show experimentally that their algorithm outperforms the state-of-the-art FPLLL [24], implementation as well as the algorithm of [22]. Koy and Schnorr [25] introduced the Segment LLL algorithm - a variant of LLL which yields a slightly weaker reduced basis but is more efficient by a factor n. This is achieved by partitioning a basis of dimension $n = km$ into m segments comprising k consecutive vectors and LLL reducing these segments. This was improved both in terms of efficiency and basis quality in [26]. The authors chose overlapping blocks in a similar way as BKZ [6] so that the global quality of the basis was improved with each "tour". Another important direction is the application of LLL to lattices with an underlying structure or form, for example, ideal lattices [27], module lattices [28] and parametric lattices [29]. It is well understood that LLL generally provides much better output quality than the analysis of the LLL-reduced bases suggests. To this end, in [30] and [31], experimental analysis have been performed on the average-case behaviour of LLL, and comparisons are drawn with the worst-case theoretical results. We note that the key ideas associated with the directions in this paragraph are more or less orthogonal to the techniques we introduce in this paper.

2 Preliminaries

Notation.

The sets of integers, rational and real numbers are denoted by $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}$ respectively. Let $[n] = \{1, \ldots, n\}$. For $x \in \mathbb{R}$, $|x|$ denotes its absolute value. The integer closest to $x \in \mathbb{R}$ is denoted by $\lfloor x \rceil$. All vectors are column vectors. The Euclidean norm of a vector $\mathbf{x} \in \mathbb{R}^m$ is denoted by $\|x\|$. The inner product of vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^m$ is denoted by $\langle \mathbf{x}, \mathbf{y} \rangle$. All logarithms are base 2 unless denoted otherwise.

Lattice, Bases, Sublattice and Linear Span.

A lattice $\mathcal{L} = \{\mathbf{Bx} : \mathbf{x} \in \mathbb{Z}^n\}$ specified by an ordered set of linearly independent vectors called a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n) \in \mathbb{R}^{m \times n}$, is denoted as $\mathcal{L}(\mathbf{B})$. We call m the dimension and n the rank of the lattice $\mathcal{L}$, where $m \geq n$. The linear span of $\mathbf{B}$ is given by $\mathtt{span}(\mathbf{B}) = \{\mathbf{Bx} : \mathbf{x} \in \mathbb{R}^n\}$. A subset of vectors in $\mathbf{B}$ gives rise to a sublattice of $\mathcal{L}(\mathbf{B})$. For example, given a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ for a lattice $\mathcal{L}$, the vectors $(\mathbf{b}_1, \ldots, \mathbf{b}_i), 1 \leq i \leq n$ form a basis of a sublattice of $\mathcal{L}$ that we denote as $\mathcal{L}_i$.

For a lattice of dimension $n \geq 2$, there are infinitely many bases. If $\mathbf{B}_1$ is a basis for a lattice $\mathcal{L}$, we may transform this into another basis $\mathbf{B}_2$ for the same lattice by $\mathbf{B}_2 = \mathbf{B}_1\mathbf{U}$, where $\mathbf{U} \in GL_n(\mathbb{Z})$ is a unimodular matrix. An invariant across the infinitely many bases of a lattice is its volume. For a basis $\mathbf{B}$ of the lattice, its volume is given by $\mathrm{Vol}(\mathcal{L}) = \sqrt{\det(\mathbf{B}^T\mathbf{B})}$ and geometrically it represents the volume of the fundamental parallelepiped of the lattice. We generally only consider lattices with vectors in $\mathbb{Q}^m$ and by scaling we need only consider lattices in $\mathbb{Z}^m$.

Gram-Schmidt Orthogonalisation.

For an ordered set of linearly independent vectors $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n), \mathbf{b}_i \in \mathbb{R}^m$, its Gram-Schmidt orthogonalisation (GSO) gives the corresponding set $\mathbf{B}^* = (\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*)$ of orthogonal vectors defined recursively as follows.

- $\mathbf{b}_1^* = \mathbf{b}_1$, and
- for $i > 1$, $\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^*$,

where a GSO coefficient $\mu_{i,j}$ is defined for $1 \leq j \leq i \leq n$ as

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

It is easy to see that $\mu_{i,i} = 1$ for all $1 \leq i \leq n$.

Orthogonal Projections.

Given a vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$, its projections $\pi_i(\mathbf{v})$ are defined for $1 \leq i \leq n$ as

- $\pi_1(\mathbf{v}) = \mathbf{v}$, and
- for $2 \leq i \leq n$, $\pi_i(\mathbf{v})$ is the projection of $\mathbf{v}$ orthogonal to $\texttt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}))$ of the sublattice $\mathcal{L}_{i-1}$.

The projection $\pi_i(\mathbf{b}_k)$ is written in terms of the GSO vectors $(\mathbf{b}_i^*, \ldots, \mathbf{b}_k^*)$ and the GSO coefficients $\mu_{k,i}, \ldots, \mu_{k,k-1}$ as follows

$$\pi_i(\mathbf{b}_k) = \mathbf{b}_k^* + \sum_{l=i}^{k-1} \mu_{k,l} \mathbf{b}_l^*.$$

In the simplest case, $\pi_i(\mathbf{b}_i) = \mathbf{b}_i^*$.

Lovász condition.

For the parameter $1/4 < \delta \leq 1$, the Lovász condition between consecutive vectors $\mathbf{b}_{k-1}, \mathbf{b}_k \in \mathbf{B}$ is defined as

$$\delta \cdot \left\| \pi_{k-1}(\mathbf{b}_{k-1}) \right\|^2 \leq \left\| \pi_{k-1}(\mathbf{b}_k) \right\|^2.$$

This can be written as $\left(\delta - \mu_{k,k-1}^2\right) \cdot \left\| \mathbf{b}_{k-1}^* \right\|^2 \leq \left\| \mathbf{b}_k^* \right\|^2$ in terms of the GSO vectors and coefficients. For all $1 \leq i < k \leq n$, the Lovász condition can be generalised (for deep insertions) as

$$\delta \cdot \left\| \pi_i(\mathbf{b}_i) \right\|^2 \leq \left\| \pi_i(\mathbf{b}_k) \right\|^2.$$

Size reduction.

Given a basis $\mathbf{B}$ for a lattice $\mathcal{L}$, size reduction of $\mathbf{b}_i$ with $\mathbf{b}_j$ replaces $\mathbf{b}_i$ with the vector $\mathbf{b}_i - \lfloor \mu_{i,j} \rceil \, \mathbf{b}_j$ while $\mathbf{b}_j$ remains unchanged. If $|\mu_{i,j}| < 1/2$, the vector $\mathbf{b}_i$ remains unchanged. Algorithm 1 describes the size reduction of a vector $\mathbf{b}_k$ with all its previous

Algorithm 1: The size reduction algorithm for a vector $\mathbf{b}_k$

Input: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, its GSO coefficients $\mu_{i,j}$, and an index k.
Output: A basis $\mathbf{B}' = (\mathbf{b_1}', \ldots, \mathbf{b}'_n)$ where $\mathbf{b}'_k$ is size reduced, and the updated coefficients $\mu'_{i,j}$.

```
for j = k − 1, ..., 1 /* The 'reverse order' as in Remark 2 */ do
    if |μ_{k,j}| > 1/2 then
        b_k ← b_k − ⌊μ_{k,j}⌉ b_j
        μ_{k,j} ← μ_{k,j} − ⌊μ_{k,j}⌉
        for i = 1, ..., j − 1 do
            μ_{k,i} ← μ_{k,i} − ⌊μ_{k,j}⌉ μ_{j,i} /* As in Remark 1 part (3) */
return B' with size reduced b'_k and updated coefficients μ'_{i,j}.
```

vectors $\mathbf{b}_{k-1}, \ldots, \mathbf{b}_1$ in the basis. The changes in the GSO coefficients $\mu_{i,j}$ due to size reduction have been described in Remark 1. Size reducing an entire basis $\mathbf{B}$ pertains to reducing each $\mathbf{b}_k$ for $2 \leq k \leq n$ with all previous vectors $\mathbf{b}_i$ for $1 \leq i < k$ in the ordering. The details are in Remark 2.

Definition 1 (Size reduced basis)**.** *A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ is said to be size reduced if for all $1 \leq j < i \leq n$, $|\mu_{i,j}| \leq 1/2$.*

We also define the notion of size-reduction where the vector $\mathbf{b}_i$ is unchanged if $|\mu_{i,j}| < \eta$ for some $\eta \geq 1/2$. This is important for implementations of size reduction where floating point approximations to real numbers are used.

Definition 2 (η-Size reduced basis)**.** *For some $\eta > 1/2$, a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ is said to be η-size reduced if for all $1 \leq j < i \leq n$, $|\mu_{i,j}| \leq \eta$.*

Remark 1 (Changes in GSO Coefficients Upon Size Reduction)**.** *Based on the descriptions in [3, Chapter 2] and [32], we know that, upon a size reduction of $\mathbf{b}_i$ with $\mathbf{b}_j$ $(1 \leq i < j)$, the values of $\mu_{i,j}$ must be updated as follows for consistency.*

1. *We set $\mu_{i,j} \leftarrow \mu_{i,j} - \lfloor \mu_{i,j} \rceil$; as a result, upon reducing $\mathbf{b}_i$ with $\mathbf{b}_j$, we get $|\mu_{i,j}| \leq 1/2$.*
2. *For $j < l < i$, the values of $\mu_{i,l}$ remain unchanged. This is based on [3] and [32, Exercise 17.4.8 (3)]. The proof is as follows[2]. Let $\mathbf{b}_i$ be already size reduced with respect to the vectors $\mathbf{b}_{i-1}, \mathbf{b}_{i-2}, \ldots, \mathbf{b}_{j+1}$. Now, we size reduce $\mathbf{b}_i$ with $\mathbf{b}_j$ to get $\mathbf{b}'_i = \mathbf{b}_i - \lfloor \mu_{i,j} \rceil \mathbf{b}_j$. Let $\mu'_{i,l}$ be the value of $\mu_{i,l}$ after the size reduction of $\mathbf{b}_i$ with respect to $\mathbf{b}_j$. Then we have*

$$\mu'_{i,l} = \frac{\langle \mathbf{b}'_i, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} = \frac{\langle \mathbf{b}_i - \lfloor \mu_{i,j} \rceil \mathbf{b}_j, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} = \frac{\langle \mathbf{b}_i, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|} - \lfloor \mu_{i,j} \rceil \frac{\langle \mathbf{b}_j, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|}.$$

Note that since $l > j$, we have $\mathbf{b}_j \perp \mathbf{b}^*_l$ as $\mathbf{b}^*_l$ is (by definition) orthogonal to $\mathbf{b}_1, \ldots, \mathbf{b}_{l-1}$. Therefore, we have $\langle \mathbf{b}_j, \mathbf{b}^*_l \rangle = 0$, and hence

$$\mu'_{i,l} = \frac{\langle \mathbf{b}_i, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} - \lfloor \mu_{i,j} \rceil \frac{\langle \mathbf{b}_j, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} = \frac{\langle \mathbf{b}_i, \mathbf{b}^*_l \rangle}{\|\mathbf{b}^*_l\|^2} = \mu_{i,l}.$$

[2] Although quite straightforward, the proof is not detailed in the literature to the best of our knowledge.

3. For all $1 \leq l \leq j-1$, *we set* $\mu_{i,l} \leftarrow \mu_{i,l} - \lfloor \mu_{i,j} \rceil \mu_{j,l}$.

In summary, if we size reduce $\mathbf{b}_i$ *with* $\mathbf{b}_j$, *then the values* $\mu_{i,l}$ *for* $j < l \leq i-1$ do not change. *However, the values* $\mu_{i,l}$ *for* $1 \leq l \leq j$ may change.

Remark 2 (Reducing in Reverse)**.** *Note that in Algorithm 1, while size reducing the vector* $\mathbf{b}_k$ *with* $\mathbf{b}_1, \ldots, \mathbf{b}_{k-1}$, *we must reduce 'in reverse'. In other words, we first reduce* $\mathbf{b}_k$ *with* $\mathbf{b}_{k-1}$, *then* $\mathbf{b}_{k-2}$, *and so on, down to* $\mathbf{b}_1$. *This is for two reasons. First, as per point (2) of Remark 1, upon size reduction of* $\mathbf{b}_k$ *with* $\mathbf{b}_i$, *the vector* $\mathbf{b}_k$ *is still size reduced with respect to all* $\mathbf{b}_l$ *for* $i < l < k$. *Second, the size reduction of* $\mathbf{b}_k$ *with* $\mathbf{b}_i$ *for* $1 \leq i < k$ *affects the size reducedness of* $\mathbf{b}_k$ *with respect to* $\mathbf{b}_l$ *for* $l < i$ *as per point (3) in Remark 1. So by size reducing* $\mathbf{b}_k$ *with* $\mathbf{b}_i$, *only the vectors before* $\mathbf{b}_i$ *are candidates for further size reduction of* $\mathbf{b}_k$ *and not the ones between* $\mathbf{b}_i$ *and* $\mathbf{b}_k$.

Lattice Reduction.

Given a basis for a lattice, the goal of lattice reduction is to transform it into a better quality basis consisting of shorter, more orthogonal vectors. Lattice reduction algorithms like LLL and its variants conduct size reduction as well as reordering of the input basis to improve their quality.

Basis Quality Measures.

Several measures can be used to describe the quality of a basis. The most widely used is the Hermite factor (HF)

$$\gamma = \frac{\|\mathbf{b}_1\|}{\mathrm{Vol}(\mathcal{L})^{1/n}}$$

of a lattice. The vector $\mathbf{b}_1$ is assumed to be the shortest vector in the output basis. It has been shown that the smaller the Hermite factor of a basis, the better the basis quality [33]. Furthermore, the *root Hermite factor* (RHF) given by $\gamma^{1/n} = \left(\frac{\|\mathbf{b}_1\|}{\mathrm{Vol}(\mathcal{L})^{1/n}}\right)^{1/n}$ can be shown experimentally [33] to converge to a constant for certain basis reduction algorithms and large n.

The potential (Pot) of a basis $\mathbf{B}$ is defined in terms of its GSO vectors $\mathbf{B}^*$ as

$$\mathrm{Pot}(\mathbf{B}) = \prod_{i=1}^{n} \mathrm{Vol}(\mathcal{L}_i)^2 = \prod_{i=1}^{n} \|\mathbf{b}_i^*\|^{2(n-i+1)}.$$

It was introduced in [1] to prove that LLL runs in polynomial time. The potential takes into account not only the vectors in a lattice basis but also their ordering. Earlier basis vectors have significantly more contribution to the value of $\mathrm{Pot}(\mathbf{B})$ than the later ones. We use the natural logarithm of the potential for easy handling of the large exponents in its computation, especially with large values of n.

$$\log_e(\mathrm{Pot}(\mathbf{B})) = \log_e\left(\prod_{i=1}^{n} \mathrm{Vol}(\mathcal{L}_i)^2\right) = 2\sum_{i=1}^{n}(n-i+1)\log_e(\|\mathbf{b}_i^*\|).$$

Another measure of basis quality is the squared sum (SS) of its GSO vectors $\mathbf{B}^*$, introduced in [7] as

$$\mathrm{SS}(\mathbf{B}) = \sum_{i=1}^{n} \|\mathbf{b}_i^*\|^2 .$$

Similarly to $\mathrm{Pot}(\cdot)$, the squared sum varies with changes in the lengths of the GSO vectors. However, unlike $\mathrm{Pot}(\cdot)$, all GSO vectors contribute equally to its value.

Ordering of Basis Vectors.

Let S_n be the group of permutations of the elements in [n]. For $\sigma \in S_n$ and a basis $\mathbf{B}$, we define $\sigma(\mathbf{B}) = \left(\mathbf{b}_{\sigma(1)}, \ldots, \mathbf{b}_{\sigma(n)}\right)$ to be a permutation of the basis vectors. Here, $\sigma(j)$ is the index of the vector in $\mathbf{B}$ that takes position j in the permuted basis $\sigma(\mathbf{B})$. In particular, we are interested in the permutations $\sigma_{i,k} \in S_n$ for $1 \leq i < k \leq n$ defined as follows.

$$\sigma_{i,k}(j) = \begin{cases} j & \text{if } \; j < i \text{ or } k < j \\ k & \text{if } \; j = i \\ j-1 & \text{if } \; i+1 \leq j \leq k. \end{cases}$$

Such a permutation of $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ essentially gives us the permuted basis

$$\sigma_{i,k}(\mathbf{B}) = (\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_k, \mathbf{b}_i, \ldots, \mathbf{b}_{k-1}, \mathbf{b}_{k+1}, \ldots, \mathbf{b}_n)$$

where $\mathbf{b}_k$ is inserted between $\mathbf{b}_{i-1}$ and $\mathbf{b}_i$, and all vectors $\mathbf{b}_i, \ldots, \mathbf{b}_{k-1}$ are shifted up by one position. The other vectors retain their positions in the ordering.

Change in Basis Quality through Permutations.

Let $X(\mathbf{B})$ be a measure of basis quality (like the HF, RHF, Pot, SS, etc.) of $\mathbf{B}$. On permuting the basis $\mathbf{B}$ to $\sigma_{i,k}(\mathbf{B})$, the difference in the measure is denoted as

$$\Delta X_{i,k} = X(\mathbf{B}) - X(\sigma_{i,k}(\mathbf{B})).$$

In particular, we get $\Delta\mathrm{Pot}_{i,k} = \mathrm{Pot}(\mathbf{B}) - \mathrm{Pot}(\sigma_{i,k}(\mathbf{B}))$ and $\Delta\mathrm{SS}_{i,k} = \mathrm{SS}(\mathbf{B}) - \mathrm{SS}(\sigma_{i,k}(\mathbf{B}))$[3]. We note that $\texttt{argmax}_{1 \leq i < k \leq n}(\Delta X_{i,k})$ returns the pair of indices (i, k) for which the value of $\Delta X_{i,k}$ is maximised.

3 The LLL Algorithm, Its Variants and Generalisations

Given a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, we have its GSO $\mathbf{B}^* = (\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*)$ and the coefficients $\mu_{i,j}$ therein.

Definition 3 (δ-LLL reduced basis)**.** *Given $1/4 < \delta \leq 1$, a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ is said to be δ-LLL reduced if the following two conditions are satisfied.*

[3]Note that even though the expression for computing the measure $\mathrm{SS}(\mathbf{B})$ itself gives equal weight to all GSO vectors (unlike $\mathrm{Pot}(\mathbf{B})$) independent of where they occur in the ordering of $\mathbf{B}^*$, the GSO vectors themselves (and hence their lengths) change upon reordering. As a result, the value of the measure $\mathrm{SS}(\mathbf{B})$ generally changes after reordering the basis.

1. $\mathbf{B}$ *is size reduced as in Definition 1.*
2. *For all* $2 \leq k \leq n$*, the Lovász condition holds between the consecutive vectors* $\mathbf{b}_{k-1}, \mathbf{b}_k \in \mathbf{B}$*. In other words,*

$$\delta \cdot \|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 \leq \|\pi_{k-1}(\mathbf{b}_k)\|^2 .$$

Algorithm 2: The LLL Algorithm [1]

Input: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, a threshold $1/4 < \delta \leq 1$
Output: A basis $\mathbf{B}' = (\mathbf{b_1}', \ldots, \mathbf{b}'_n)$ which is δ-LLL reduced

```
Find the GSO basis B* and initialise the values of μ_{i,j}
k ← 2
while k ≤ n do
    Size reduce b_k /* As in Algorithm 1 */
    if ‖b*_k‖² < (δ − μ²_{k,k−1}) ‖b*_{k−1}‖² /* Equivalent to the failure of the
     condition in (1) */ then
        B ← σ_{k−1,k}(B) /* Swap vectors b_{k−1}, b_k ∈ B */
        Update b*_{k−1}, b*_k /* As in [32, Lemma 17.4.3] */
        Update μ_{i,j}'s /* As in [3, Algorithm 2.6.3] */
        k ← max(k − 1, 2)
    else
        k ← k + 1
return B′, a δ-LLL reduced basis
```

Definition 4 (δ-DeepLLL reduced basis)**.** *Given* $1/4 < \delta \leq 1$*, a basis* $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ *is said to be* δ-DeepLLL *reduced if the following two conditions are satisfied.*

1. $\mathbf{B}$ *is size reduced as in Definition 1.*
2. *For all* $1 \leq i < k \leq n$*,*

$$\delta \cdot \|\pi_i(\mathbf{b}_i)\|^2 \leq \|\pi_i(\mathbf{b}_k)\|^2 .$$

Remark 3. *If the Lovász condition holds for* all *pairs* (i, k)*, then it must certainly hold for all consecutive pairs* $(k-1, k)$*. A* δ-DeepLLL *reduced basis is hence* δ-LLL *reduced.*

The LLL and DeepLLL Algorithms.

The LLL algorithm [1] is described in Algorithm 2. The output basis $\mathbf{B}'$ is δ-LLL reduced as in Definition 3. A swap between vectors $\mathbf{b}_{k-1}$ and $\mathbf{b}_k$ in the algorithm is denoted by $\mathbf{B} \leftarrow \sigma_{k-1,k}(\mathbf{B})$. This is generalised in DeepLLL [6] and its variants [9, 7] to a deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ where $1 \leq i < k \leq n$. All our descriptions are in terms of deep insertions. The corresponding results for swaps can be derived by substituting $i = k - 1$, where applicable.

Remark 4 (Measure for Lovász Condition)**.** *The Lovász condition is given by* $\delta \cdot \|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 \leq \|\pi_{k-1}(\mathbf{b}_k)\|^2$*. This can be written as*

$$(1-\delta) \cdot \|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 \geq \|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 - \|\pi_{k-1}(\mathbf{b}_k)\|^2 .$$

Here, $\|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 - \|\pi_{k-1}(\mathbf{b}_k)\|^2$ *denotes the change in* $\|\pi_{k-1}(\mathbf{b}_{k-1})\|^2 = \left\|\mathbf{b}^*_{k-1}\right\|^2$ *(the square of the length of the* $(k-1)^{th}$ *GSO vector) that will occur if a swap step* $\mathbf{B} \leftarrow \sigma_{k-1,k}(\mathbf{B})$ *was to happen. In Algorithm 2, if the condition is not satisfied, then the change in* $\left\|\mathbf{b}^*_{k-1}\right\|^2$ *is large enough to go ahead with the swap and bring vector* $\mathbf{b}_k$ *to the earlier position* $k-1$ *in the basis ordering. In general, for* $1 \leq i < k \leq n$*, the Lovász condition for a deep insertion* $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ *is given by* $\delta \cdot \|\pi_i(\mathbf{b}_i)\|^2 \leq \|\pi_i(\mathbf{b}_k)\|^2$ *which can also be written similarly as*

$$(1-\delta) \cdot \|\pi_i(\mathbf{b}_i)\|^2 \geq \|\pi_i(\mathbf{b}_i)\|^2 - \|\pi_i(\mathbf{b}_k)\|^2 .$$

Based on the above, we observe that the (generalised) Lovász condition essentially uses a localised measure of the quality of the basis. For an index $1 \leq i < n$, the measure of quality of the basis $\mathbf{B}$ is given by $\mathrm{LC}_i(\mathbf{B}) = \|\pi_i(\mathbf{b}_i)\|^2 = \|\mathbf{b}^*_i\|^2$*. The change* $\Delta\mathrm{LC}_i$ *in the quality of the basis due to a deep insertion* $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ *is given by*

$$\Delta\mathrm{LC}_i = \mathrm{LC}_i(\mathbf{B}) - \mathrm{LC}_i(\sigma_{i,k}(\mathbf{B})) = \|\pi_i(\mathbf{b}_i)\|^2 - \|\pi_i(\mathbf{b}_k)\|^2 .$$

Then, the generalised Lovász condition can be written as

$$(1-\delta) \cdot \mathrm{LC}_i(\mathbf{B}) \geq \Delta\mathrm{LC}_i \tag{1}$$

which fails if $\Delta\mathrm{LC}_i > (1-\delta) \cdot \mathrm{LC}_i(\mathbf{B})$ *and calls for deep insertion. This interpretation of the Lovász condition as a change in the measure of basis quality is not present in the literature to the best of our knowledge.*

Thus the condition of the `if` *statement in step* 8 *of Algorithm 3 is a further generalisation of the generalised Lovász condition for any measure of quality* $X(\mathbf{B})$ *of the basis* $\mathbf{B}$*. In Algorithm 3, if the condition is not satisfied, bringing a later vector* $\mathbf{b}_k$ *to an earlier position* i *in the basis ordering will result in appreciable improvement in the basis quality* $X(\mathbf{B})$*.*

Variants of DeepLLL: transition from a local measure to a global measure of quality

As noted above, the Lovász condition in LLL [1] and its generalisation in DeepLLL [6] are both used to check the decrease in $\|\pi_i(\mathbf{b}_i)\| = \|\mathbf{b}^*_i\|$ by inserting a later vector $\mathbf{b}_k$ at an earlier position $i < k$. The length of a GSO vector is a localised measure of quality that does not capture the quality of the whole basis. This changed in Pot-DeepLLL [9] where instead of a localised measure of quality, the potential $\mathrm{Pot}(\cdot)$ was used in DeepLLL so that the effect of permuting vectors on the entire basis is considered. In SS-DeepLLL [7], $\mathrm{Pot}(\cdot)$ was replaced by another global measure $\mathrm{SS}(\cdot)$.

The basic operation of deep insertion for reordering the basis vectors is the same in all three algorithms.

Definition 5 (δ-X-DeepLLL reduced basis)**.** *Given $0 < \delta \leq 1$, a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ is said to be δ-X-DeepLLL reduced for a basis quality measure $X(\cdot)$ if the following two conditions are satisfied.*

1. $\mathbf{B}$ *is size reduced as in Definition 1.*
2. *For all $1 \leq i < k \leq n$,*
$$\delta \cdot X(\mathbf{B}) \leq X(\sigma_{i,k}(\mathbf{B})).$$

We omit the δ in naming our algorithms. Unless an algorithm is run for two different values of δ, this parameter is an implicit input to the algorithm. The choice of δ is however crucial in determining the quality of the basis. Larger the value of δ, the better the output quality in general. Hence we include it in the notation used in the definition of reducedness of a basis.

Using basis quality measures $\text{Pot}(\cdot)$ and $\text{SS}(\cdot)$ in place of the generic $X(\cdot)$, Definition 5 is instantiated to that of a Pot-DeepLLL [9] reduced basis and a SS-DeepLLL [7] reduced basis. We know from [9, Lemma 2] that a Pot-DeepLLL reduced basis is LLL reduced. Also from [9, Lemma 3], for $1/4^{n-1} < \delta \leq 1$, a δ-DeepLLL reduced basis is δ^{n-1}-Pot-DeepLLL reduced. From [7, Proposition 1] we know that any 1-SS-DeepLLL reduced basis is also δ-LLL reduced for any $1/4 < \delta < 1$. However, there are no known relationships between δ-SS-DeepLLL reduced bases and δ-DeepLLL reduced bases to the best of our knowledge. We remark that both Pot-DeepLLL and SS-DeepLLL have polynomial-time complexity by construction, but their output quality cannot be covered by [7, Theorem 1] since their output bases are not DeepLLL-reduced.

Remark 5. *In general, for two different basis quality measures X_1 and X_2, a δ-X_1-*DeepLLL *reduced basis may or may not be δ-X_2-*DeepLLL *reduced. In particular, a δ-*Pot-DeepLLL *reduced basis is also δ-*LLL *reduced whilst a δ-*SS-LLL *reduced basis is not necessarily so for $\delta < 1$. Therefore, there exist bases which are δ-*SS-LLL *reduced but not necessarily δ-*LLL *reduced or δ-*Pot-DeepLLL *reduced.*

A Generalisation of DeepLLL, Pot-DeepLLL and SS-DeepLLL.

We provide a generalised description of DeepLLL and its variants Pot-DeepLLL and SS-DeepLLL in the X-DeepLLL algorithm 3. The X in the name X-DeepLLL corresponds to the general measure $X(\mathbf{B})$ of the quality of $\mathbf{B}$. The generalisation is instantiated for different local and global quality measures of a basis $\mathbf{B}$ that are all based on the GSO vectors $\mathbf{B}^*$. The localised measure $\text{LC}_i(\mathbf{B}) = \|\mathbf{b}_i^*\|^2$ (used in DeepLLL [6]) is only for a single GSO basis vector, while the measures $\text{Pot}(\cdot)$ [9] and $\text{SS}(\cdot)$ [7] are on the entire GSO basis $\mathbf{B}^*$. In Remark 4 we have argued that the (generalised) Lovász condition can be interpreted as a condition on the change in the quality of the basis assessed based on the localised measure $\text{LC}_i(\mathbf{B}) = \|\pi_i(\mathbf{b}_i)\|^2 = \|\mathbf{b}_i^*\|^2$. Hence, the X-DeepLLL algorithm 3 is a generalisation of the DeepLLL algorithm of [6]. Pot-DeepLLL and SS-DeepLLL are both variants of DeepLLL. It should be easy to see that the X-DeepLLL algorithm 3 for the measure $X(\cdot) = \text{Pot}(\cdot)$ is Pot-DeepLLL and for the measure $X(\cdot) = \text{SS}(\cdot)$, it is SS-DeepLLL.

Algorithm 3: The X-DeepLLL Algorithm

Input: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, a threshold $0 < \delta \leq 1$
Output: $\mathbf{B}' = (\mathbf{b_1}', \ldots, \mathbf{b}_n')$ which is δ-X-DeepLLL reduced

Find the GSO basis $\mathbf{B}^*$ and initialise the values of $\mu_{i,j}$
Size reduce $\mathbf{b}_2, \ldots, \mathbf{b}_n$ /* As in Algorithm 1 */
Initialise other bookkeeping data structures, if required for $X(\mathbf{B})$
$k \leftarrow 2$
while $k \leq n$ **do**
    Size reduce $\mathbf{b}_k$ /* As in Algorithm 1 */
    Find i such that $i = \texttt{argmax}_{1 \leq j < k}(\Delta X_{j,k})$ and set $\Delta X = \Delta X_{i,k}$
    if $\Delta X > (1 - \delta) \cdot X(\mathbf{B})$ **then**
        $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ /* Deep insert $\mathbf{b}_k$ before $\mathbf{b}_i$ */
        Update $\mathbf{B}^*$ and $\mu_{l,j}$ /* As in [11, Theorem 1 and Proposition 1] */
        $k \leftarrow \max(i, 2)$
    else
        $k \leftarrow k + 1$
return $\mathbf{B}'$, a δ-X-DeepLLL reduced basis

In the generalised X-DeepLLL algorithm 3, we note that the threshold value δ represents a fraction of the measure $X(\mathbf{B})$. If a reordering of the basis $\mathbf{B}$ can improve its quality by *more than* $(1-\delta) \cdot X(\mathbf{B})$, the algorithm has scope for such a reordering. In fact, when there is no way to decrease the measure (thus improving the quality) to less than the fraction $\delta \cdot X(\mathbf{B})$ of the measure, that is when the basis is considered to be δ-X-DeepLLL reduced as in Definition 4. As may be expected, the threshold δ depends on the measure X in the context. The notation δ is commonly used [1, 6, 9] to denote the fraction in the context of algorithms based on the localised measure $\mathrm{LC}(\cdot)$ (when using the Lovász condition) and the measure $\mathrm{Pot}(\cdot)$ for the whole basis. The notation η has been used in [7] to denote the threshold in the context of the measure $\mathrm{SS}(\cdot)$. In our generalisations of the algorithms and their analysis, we continue using the more common notation δ with the awareness that for two different measures $X_1(\cdot)$ and $X_2(\cdot)$, two different thresholds δ_1 and δ_2 may have to be considered, respectively. The relationship between threshold values δ_1, δ_2 of the algorithms may be derived from the relationship between their measures X_1, X_2 as in [9, 7].

It should be clear that the value of the threshold δ in the X-DeepLLL algorithm 3 should be upper bounded by $\delta \leq 1$ (and consequently $(1 - \delta) \geq 0$) due to the algorithm's key principle of trying to reduce the measure $X(\mathbf{B})$ in every iteration as explained above. In particular, for $\delta = 1$, a deep insertion is allowed for any decrease $\Delta X > 0$ in the measure. Assuming the measure $X(\mathbf{B}) > 0$ for any basis $\mathbf{B}$, since the decrease in the measure ΔX can not be more than or equal to the measure $X(\mathbf{B})$ itself, hence we necessarily have $\delta > 0$. For the algorithms using the measure $\mathrm{LC}(\cdot)$ (based on the Lovász condition), the threshold must further satisfy $\delta > 0.25$ [1]. In general, the threshold δ and a tighter lower bound thereof may be determined by the termination condition for the loop.

The LLL-style algorithms mentioned above (LLL, DeepLLL, SS-DeepLLL, Pot-DeepLLL) work in a manner where a single iteration of the loop works only with a sublattice $\mathcal{L}_k$ generated by $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ of $\mathbf{B}$. The rest of the vectors $(\mathbf{b}_{k+1}, \ldots, \mathbf{b}_n)$ remain "untouched" in that iteration. Hence, after a deep insertion step, $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ the sublattice under consideration in the next iteration would just be $(\mathbf{b}_1, \ldots, \mathbf{b}_i)$. The newly inserted vector $\mathbf{b}_i$ will have already been size reduced with respect to the vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}$ in the previous iteration of the loop when considering the index k. The vectors $\mathbf{b}_{i+1}, \ldots, \mathbf{b}_k$ have all been "shifted" up by one position. They will now require further size reduction since they will not have been reduced with respect to the newly inserted vector $\mathbf{b}_i$. However, this does not need to be done immediately; these vectors will be size reduced again when they enter the sublattice under consideration in a subsequent iteration of the loop. So these algorithms only size reduce one vector in an iteration and not the whole basis.

Deep Insertions.

A deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ only changes the vectors $\mathbf{b}_i, \ldots, \mathbf{b}_k$ in the basis. The vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_{k+1}, \ldots, \mathbf{b}_n$ remain unchanged. The corresponding changes in the GSO basis $\mathbf{B}^*$ and the lengths of the vectors therein are given by [11, Theorem 1]. The corresponding changes in the GSO coefficients are given by [11, Proposition 1].

Remark 6. *From [11, Theorem 1], we note that due to a deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ the only GSO vectors that change are $\mathbf{b}_i^*, \ldots, \mathbf{b}_k^*$. Hence, the only GSO coefficients that change are $\mu_{l,j}$ for $j < l$, $i \leq j \leq k$, and $i + 1 \leq l \leq n$.*

Algorithm 4: The X-GGLLL Algorithm

Input: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$, a threshold $0 < \delta \leq 1$
Output: $\mathbf{B}' = (\mathbf{b_1}', \ldots, \mathbf{b}_n')$ which is a δ-X-GGLLL reduced basis

Find the GSO basis $\mathbf{B}^*$ and initialise the values of $\mu_{i,j}$
Size reduce $\mathbf{b}_2, \ldots, \mathbf{b}_n$ in this order /* As in Algorithm 1 for each $\mathbf{b}_k$ */
Find (i', k') such that $(i', k') = \texttt{argmax}_{1 \leq i < k \leq n}(\Delta X_{i,k})$ and set $\Delta X = \Delta X_{i',k'}$
while $\Delta X > (1 - \delta) \cdot X(\mathbf{B})$ **do**
    $\mathbf{B} \leftarrow \sigma_{i',k'}(\mathbf{B})$ /* Deep insert $\mathbf{b}_{k'}$ before $\mathbf{b}_{i'}$ */
    Update $\mathbf{B}^*$ and $\mu_{l,j}$ /* As in [11, Theorem 1 and Proposition 1] */
    Size reduce $\mathbf{b}_{i'+1}, \ldots, \mathbf{b}_n$ /* As in Algorithm 1 and proof of Lemma 1*/
    Find (i', k') such that $(i', k') = \texttt{argmax}_{1 \leq i < k \leq n}(\Delta X_{i,k})$ and $\Delta X = \Delta X_{i',k'}$
end
return $\mathbf{B}'$, a δ-X-DeepLLL reduced basis.

4 The X-GGLLL Algorithm

The generalisation of DeepLLL, Pot-DeepLLL and SS-DeepLLL in the form of the X-DeepLLL algorithm 3 sets the stage for our new framework of algorithms.

The Greedy Global Framework.

The greedy global framework described as the X-GGLLL algorithm 4 provides a general description of algorithms realised by specifying a basis quality measure X. The algorithm starts by finding the GSO in step 1 and then size reducing the input basis $\mathbf{B}$ in step 2. In step 3, it finds a pair of indices (i', k') that may be suitable for a deep insertion $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$. It then runs a loop performing a deep insertion and associated bookkeeping in steps 5-6, the consequent size reductions using Algorithm 1 in step 7 and finding an appropriate pair (i', k') for the next iteration in step 8. By the end of each iteration of the loop, the algorithm produces a size reduced basis, and the associated bookkeeping information like the values of $\mu_{i,j}$, etc. are all updated to be consistent with the new basis. The loop runs as long as there is a pair of indices (i', k') such that if $\mathbf{b}_{k'}$ is deep inserted before $\mathbf{b}_{i'}$, the change in the measure $\Delta X = X(\mathbf{B}) - X(\sigma_{i',k'}(\mathbf{B}))$ is at least a fraction $(1-\delta)$ of the current measure $X(\mathbf{B})$. Note that every time the loop runs, a deep insertion is certainly conducted. For δ close to 1, $(1-\delta)$ is a small value. So the algorithm essentially terminates when there is no possible deep insertion step in the entire basis $\mathbf{B}$ that can reduce the measure $X(\mathbf{B})$ by a fraction $(1-\delta)$ that may be considered as a substantial change to the quality of the basis. Thus X-GGLLL returns a δ-X-DeepLLL reduced basis as in Definition 5. We prove this in Lemma 2.

In a single iteration, LLL, DeepLLL, Pot-DeepLLL and SS-DeepLLL either increase the index $2 \leq k \leq n$ by 1, or decrease the index by some value $i < k-1$. In the process, they only work with the sublattice $\mathcal{L}_k$ generated by a subset $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ of $\mathbf{B}$. The key novelty of our framework and the algorithms therefrom lies in not restricting the choice of the vector $\mathbf{b}_k$ that is investigated for a possible insertion at an earlier position to only a sublattice (unlike all previous LLL-style algorithms). Instead, our algorithm works with the whole basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ and hence the entire lattice in every iteration throughout the algorithm. As a result, a deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ in our algorithm has to be immediately followed by reductions of $\mathbf{b}_{i+1}, \ldots, \mathbf{b}_n$ to ensure that the entire basis is size reduced and ready for the next iteration. Even though this is $\mathcal{O}(n)$ more operations than that of X-DeepLLL, it creates avenues for smarter choices of the indices for size reduction. In asymptotic terms, this loss is compensated by the $\mathcal{O}(n)$ gain for not having to increment the index k for each deep insertion $\mathcal{O}(n)$ times in the worst case.

Apart from working with the whole basis in every iteration, we introduce a greedy technique to select the indices (i, k). In particular, the algorithm finds a pair (i', k') such that the consequent change in the measure $\Delta X(\mathbf{B})$ is maximised. Step 3 of Algorithm 4 does this for the first time before entering the loop and step 8 does it subsequently for each iteration of the loop. The algorithm starts with a certain value of $X(\mathbf{B})$ that can be at most I_X and attempts to reach a minimum value Z_X. By choosing the maximum decrease in each step, it gets closer to Z_X by reaching a δ-X-DeepLLL state very quickly (if not the quickest[4]) by taking the largest possible leaps at each point. The asymptotic analysis assumes the least possible change in the measure in every iteration and hence does not capture the effect of the greedy choice.

[4]It is well known that an immediate greedy choice is not necessarily always the best in terms of the overall result of an algorithm.

However, the gains due to the greedy choice gets reflected in the experimental results provided in Section 6 where our algorithms perform exceedingly well in terms of their runtimes, the total number of deep insertions, and total number of size reductions with respect to previous LLL-style algorithms.

The greedy choice is not necessarily the best long-term choice though. There could be other pairs (i, k) in an iteration that do not decrease the measure as much as the greedy choice (i', k') (but more than δ fraction) in that iteration, but creates the scope for a larger decrease in the measure in subsequent iterations. We do not consider such strategies in this work and leave them for future consideration. Our focus is on the greedy choice only.

Every new measure gives us a unique new lattice reduction algorithm. For the potential, we get Pot-GGLLL and for the squared sum, we get SS-GGLLL. Like X-DeepLLL, the values of δ to be used to get output bases of sufficiently good quality will depend on the measure X in the context. We assume that any other measure X will be calculable from the basis vectors and the GSO information. If necessary, the steps in Algorithm 4 can be modified to take into account possible additional bookkeeping steps that a measure may require if it is not calculable from the stored information. We note that the change in the measure X may require additional computation; for instance, the change in potential requires the calculation of projections[5]. However, these computations can be done on the fly, and are covered by step 8 of Algorithm 4.

Note that $\text{Pot}(\mathbf{B})$ and $\text{SS}(\mathbf{B})$ are global measures of quality of the basis. Given that the local measure $\text{LC}_i(\mathbf{B}) = \|\pi_i(\mathbf{b}_i)\|^2 = \|\mathbf{b}_i^*\|^2$ is on a single GSO vector, and not on the entire basis, more careful consideration must be taken to establish a greedy approach on independent $\text{LC}_i(\mathbf{B})$ measures for different indices i. So we only instantiate Algorithm 4 using global measures in this work.

Remark 7 (Preprocessing Reduction)**.** *The description of* Pot-DeepLLL *in [9, Algorithm 1] includes a preprocessing of the basis* $\mathbf{B}$ *by* LLL*. In the case of the* SS-DeepLLL *algorithm in [7, Algorithm 2], the description itself does not include the preprocessing step. However, they have included the preprocessing step with* 0.99-LLL *(that is* LLL *with* $\delta = 0.99$*) while reporting the performance results [7, Section 4.3.3]. We note here from [7] that the quality of the output basis from a reduction algorithm is often key to their subsequent use in other algorithms for finding short vectors in the lattice. Furthermore, any lattice reduction algorithm can be used for preprocessing the basis before being fed into a second algorithm for further reduction. Given that the efficiency of our algorithms (especially* SS-GGLLL*), are in practice almost as good as* LLL *in many cases, while providing much better output quality, we believe the preprocessing can be done using any algorithm that would be suitable in the context depending on an efficiency versus output quality trade-off for the given input parameters, basis types, etc. Hence, we have excluded the preprocessing step from our theoretical descriptions and asymptotic analysis of the* LLL*-style algorithms and have focused on their independent performances. In our experiments, however, we have examined the standalone algorithms as well as the algorithms after tha basis has been* 0.99-LLL *preprocessed.*

[5] In Pot-DeepLLL, when computing the position i for deep inserting a vector $\mathbf{b}_k$, it is necessary to compute the projections $\pi_i(\mathbf{b}_k)$ to check if the insertion is viable. However, this is not essential in the computation of the measure SS since the change in SS due to insertion can be computed directly using the GSO information that was updated in a previous step without computing a projection [7, Equation 5].

The BKZ *algorithm [6] runs* LLL *as a preprocess before applying the blockwise reduction. We use the FPLLL [24] library implementation of* BKZ *that inherits this feature in our experiments.*

5 Theoretical Results

Lemma 1. *Let $X(\mathbf{B})$ be lower bounded by $Z_X > 0$. Algorithm 4, outputs a size reduced basis as in Definition 1.*

Proof. In every iteration of Algorithm 4, the measure decreases by $\Delta X(\mathbf{B}) = X(\mathbf{B}) - X(\sigma_{i,k}(\mathbf{B}))$. Since it can keep decreasing only until $Z_X > 0$, the algorithm terminates and outputs a basis.

To prove that the output basis is size reduced, it is sufficient to show that the basis vectors are all size reduced by the end of each iteration of the `while` loop in Algorithm 4. We prove this by induction on the number of loop iterations. We note that step 2 of Algorithm 4 size reduces the whole basis before the first iteration. In general, we assume that the basis is size reduced at the start of iteration r. By Remark 6, a deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ only changes the GSO vectors $\mathbf{b}_i^*, \ldots, \mathbf{b}_k^*$. From [11, Theorem 1, Proposition 1] and point (2) of Remark 1, we know the following.

- *The vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}$ do not need further size reduction.* In fact, the vectors $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}$ have not changed. Since their orders have not changed either, their GSO vectors also remain the same.
- *The vector $\mathbf{b}_k$ upon being inserted in position i does not need further size reduction.* In the deep insertion step $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$, the vector $\mathbf{b}_k$ is inserted in position i. This vector has already been size reduced with respect to $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}$ in a previous iteration $< r$ (or before the loop starts). However, its GSO changes from $\pi_k(\mathbf{b}_k)$ to $\pi_i(\mathbf{b}_k)$ due to the reordering.
- *Vectors $\mathbf{b}_{i+1}, \ldots, \mathbf{b}_k$ need to be size reduced by all earlier vectors, but $\mathbf{b}_{k+1}, \ldots, \mathbf{b}_n$ need only to be reduced by $\mathbf{b}_k, \ldots \mathbf{b}_1$.* By Remark 6, the only GSO coefficients that change upon a deep insertion $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ are $\mu_{l,j}$ for $j < l$, $i \leq j \leq k$, and $i + 1 \leq l \leq n$.
  - In particular, for vectors $\mathbf{b}_l, i + 1 \leq l \leq k$, the following things change.
    - The GSO of vector $\mathbf{b}_l$ changes from being a projection of $\mathbf{b}_l$ orthogonal to $\mathtt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_{l-1}))$ to being orthogonal to $\mathtt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_k, \mathbf{b}_{i+1}, \ldots, \mathbf{b}_{l-1}))$.
    - Also, $\mathbf{b}_l$ may not be size reduced with respect to this newly inserted vector $\mathbf{b}_k$.

    Hence, we start with $\mathbf{b}_{i+1}$ and size reduce it with the newly inserted vector $\mathbf{b}_k$. Upon this size reduction, for $1 \leq l < k$, the values of $\mu_{i+1,l}$ will be updated by part (3) of Remark 1. Hence, we must size reduce $\mathbf{b}_{i+1}$ with all vectors $\mathbf{b}_i, \ldots, \mathbf{b}_1$ as explained in Remark 2. We similarly reduce all vectors $\mathbf{b}_{i+2}, \ldots, \mathbf{b}_k$.
  - Reordering the vectors $(\mathbf{b}_1, \ldots, \mathbf{b}_k)$ does not change $\mathtt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_k))$. The vectors $\mathbf{b}_{k+1}, \ldots, \mathbf{b}_n$ have not been changed due to the deep insertion step. Hence, their projections orthogonal to $\mathtt{span}((\mathbf{b}_1, \ldots, \mathbf{b}_k))$ remain the same. Thus their

GSO remains the same. Furthermore, the vectors $\mathbf{b}_{k+1}, \ldots, \mathbf{b}_n$ may not be size reduced with respect to $\mathbf{b}_{i+1}, \ldots, \mathbf{b}_k$. Therefore, vectors $\mathbf{b}_{k+1}, \ldots, \mathbf{b}_n$ must be size reduced only with the vectors $\mathbf{b}_k, \ldots, \mathbf{b}_1$ as in Remark 2.

We therefore must reduce $\mathbf{b}_{i'+1}, \ldots \mathbf{b}_n$*, due to the change in GSO of the vector in position* i'*. This is done in step* 7 *of Algorithm 4.*

□

Lemma 2. *Algorithm 4 returns a* δ*-*X*-*DeepLLL *reduced basis as in Definition 5.*

Proof. Algorithm 4 outputs a basis $\mathbf{B}'$. The condition in the `while` statement in step 4 of the algorithm ensures that upon the termination of the algorithm, no possible reordering $\sigma_{i,k}(\mathbf{B}')$ for all $1 \leq i < k \leq n$ results in a $\Delta X = X(\mathbf{B}') - X(\sigma_{i,k}(\mathbf{B}'))$ which is greater than $(1-\delta) \cdot X(\mathbf{B}')$. In other words,

$$\Delta X = X(\mathbf{B}') - X(\sigma_{i,k}(\mathbf{B}')) \leq (1-\delta) \cdot X(\mathbf{B}')$$

for all $1 \leq i < k \leq n$. Equivalently, $\delta \cdot X(\mathbf{B}') \leq X(\sigma_{i,k}(\mathbf{B}'))$ for all $1 \leq i < k \leq n$. Also by Lemma 1, the basis $\mathbf{B}'$ on output is size reduced. Hence, the output of X-GGLLL is a δ-X-DeepLLL reduced basis as per Definition 5. □

In Algorithm 4, the basis quality measure $X(\mathbf{B})$ being a function of the basis $\mathbf{B}$, may be computed using the values of the associated parameters like $\|\mathbf{b}_i\|^2$, $\|\mathbf{b}_i^*\|^2$ and $\mu_{i,k}$, for all $1 \leq i \leq k \leq n$. Let C be an upper bound on the square of the norm of the vectors in $\mathbf{B}$. The following result is on the computational complexity of the general X-GGLLL algorithm.

Lemma 3. *Let* $\|\mathbf{b}_i\|^2 \leq C$ *for all* $1 \leq i \leq n$ *in a basis* $\mathbf{B}$*. In Algorithm 4[Step 8], let the number of bit operations required for finding the pair of indices* (i', k') *for the maximum* $\Delta X_{i,k}$ *be* $\mathcal{O}(f_X(C, m, n))$ *using exact* $\mathbb{Q}$ *arithmetic but without fast integer arithmetic. Let* I_X *and* Z_X *respectively denote upper and lower bounds on* $X(\mathbf{B})$ *and let* $0 < \delta < 1$[6]*. Then the total number of bit operations performed by the* X-GGLLL *algorithm 4 is given by*

$$\mathcal{O}\left(\left(n^4 \log^2 C + mn^4 \log^2 C + f_X(C, m, n)\right) \log_{1/\delta}\left(\frac{I_X}{Z_X}\right)\right) \tag{2}$$

using exact $\mathbb{Q}$ *arithmetic but without fast integer arithmetic.*

Proof. We assume all arithmetic operations in Algorithm 4 are using exact $\mathbb{Q}$ arithmetic but without fast integer arithmetic. We first note that the size reduction step within the `while` loop ensures that the length of the vectors in the basis $\mathbf{B}$ do not increase throughout the algorithm [1]. All arithmetic operations are on integers of size $O(n \log C)$ bits by the same argument as in [1][Proposition 1.26].

[6] We note that Algorithm 4 can work with $\delta = 1$ because it can (theoretically) allow very small changes in the measure X. However, an arbitrarily small change in the measure cannot be captured by a fixed value of δ in the expression for the number of iterations. Hence, $\delta < 1$ in the analysis.

At each iteration of the `while` loop, we reduce the measure X by a factor of at least δ. So after i iterations, the measure $X^{(i)} = \frac{1}{\delta^i} X$ satisfies

$$Z_X \leq \frac{1}{\delta^i} X \leq I_X.$$

For a given δ, the iteration number i is maximised for $Z_X = \frac{1}{\delta^i} X$. Thus, the number of deep insertions (iterations of the `while` loop) is bounded above by

$$\log_{1/\delta} \left(\frac{I_X}{Z_X} \right).$$

Within each iteration of the `while` loop, we do the following operations.

- *Deep insertions*: A deep insertion and its associated bookkeeping are done in steps 5-6 of the algorithm. This results in updates of the basis parameters like the GSO vectors and the GSO coefficients. We know from the analysis of [11, Algorithm 4] that the total bit-complexity of the GSO updates is $\mathcal{O}\left(n^4 \log^2 C\right)$.
- *Size reductions*: Step 7 of the algorithm size reduces the basis and performs the associated bookkeeping updates. A vector in the basis contains m integers each of size $\mathcal{O}(n \log C)$. So the size reduction of a vector $\mathbf{b}_k$ with $\mathbf{b}_i, 1 \leq i < k$ requires $\mathcal{O}\left(mn^2 \log^2 C\right)$ bit operations. So the size reduction of $\mathbf{b}_k$ with all such $\mathbf{b}_i$ as in Algorithm 1 requires $\mathcal{O}\left(mn^3 \log^2 C\right)$ bit operations. Hence size reducing all basis vectors will require $\mathcal{O}\left(mn^4 \log^2 C\right)$ bit operations.
- *Index search*: In step 8, we search for the pair of indices (i', k') for which the measure $\Delta X_{i',k'}$ is the minimum among all possible pairs. We assume this step requires $\mathcal{O}(f_X(C, m, n))$ bit operations in every iteration.

So Algorithm 4 needs a total of $\mathcal{O}\left(n^4 \log^2 C + mn^4 \log^2 C + f_X(C, m, n)\right)$ bit operations in each iteration of the `while` loop. Considering all iterations, the total number of bit operations performed by the algorithm is given by

$$\mathcal{O}\left(\left(\underbrace{n^4 \log^2 C}_{\text{deep insertion}} + \underbrace{mn^4 \log^2 C}_{\text{size reduction}} + \underbrace{f_X(C, m, n)}_{\text{index search}} \right) \underbrace{\log_{1/\delta} \left(\frac{I_X}{Z_X} \right)}_{\#\text{iterations}} \right).$$

□

The proof of Lemma 3 shows that the asymptotic complexity of Algorithm 4 does not capture the *value* by which the measure X decreases in each iteration. Any deep insertion strategy which decreases X by a fraction at least $(1 - \delta)$ will result in an algorithm with asymptotic complexity at most as in 2 of Lemma 3. In practice, the greedy choice of an insertion that results in the maximum possible decrease in the measure makes the algorithm very efficient.

We use Lemma 3 corresponding to the general framework to find the computational complexities of the concrete algorithms Pot-GGLLL and SS-GGLLL.

5.1 Computational Complexity of Pot-GGLLL

With $\text{Vol}(\mathcal{L}_i)^2 = \prod_{j=1}^{i} \left\|\mathbf{b}_j^*\right\|^2$, the potential is given by

$$\text{Pot}(\mathbf{B}) = \prod_{i=1}^{n} \text{Vol}(\mathcal{L}_i)^2 = \prod_{i=1}^{n-1} \text{Vol}(\mathcal{L}_i)^2 \cdot \text{Vol}(\mathcal{L})^2.$$

From [9, Proof of Proposition 1] we know that an upper bound on the value of $\text{Pot}(\mathbf{B})$ is

$$I_{\text{Pot}} = \prod_{i=1}^{n-1} \text{Vol}(\mathcal{L}_i)^2 \text{Vol}(\mathcal{L})^2 \leq \prod_{i=1}^{n-1} C^i \text{Vol}(\mathcal{L})^2 \leq \prod_{i=1}^{n-1} C^{\frac{n(n-1)}{2}} \text{Vol}(\mathcal{L})^2$$

and a lower bound is $Z_{\text{Pot}} \geq \text{Vol}(\mathcal{L})^2$. In 2, we substitute the expressions for the number of iterations and the complexity of index search to find the overall complexity of Pot-GGLLL. From Lemma 3, the maximum number of iterations in Pot-GGLLL is

$$\log_{1/\delta}\left(\frac{I_{\text{Pot}}}{Z_{\text{Pot}}}\right) = \log_{1/\delta}\left(C^{n(n-1)/2}\right) = \mathcal{O}(n^2 \log_{1/\delta} C).$$

For a pair (i,k) of indices, computing the value of $\Delta\text{Pot}_{i,k}$ as described in [9, Equation 3.1] requires $\mathcal{O}(n^2)$ arithmetic operations or equivalently $\mathcal{O}(n^4 \log^2 C)$ bit operations. A straight-forward extension of this to find $\texttt{argmax}_{1\leq i<k\leq n}(\Delta\text{Pot}_{i,k})$ would require the computation of $\Delta\text{Pot}_{i,k}$ for each pair (i,k) with a total of $\mathcal{O}(n^6 \log^2 C)$ bit operations. This computation can be improved by $\mathcal{O}(n)$ time. For a fixed index k, the values of $\Delta\text{Pot}_{i,k}$ can be computed incrementally for all $i \in \{k-1,\ldots,1\}$ where $\Delta\text{Pot}_{i-1,k}$ is computed using the value of $\Delta\text{Pot}_{i,k}$. This optimisation applies to Pot-DeepLLL as well as Pot-GGLLL. Hence Pot-GGLLL computes $\texttt{argmax}_{1\leq i<k\leq n}(\Delta\text{Pot}_{i,k})$ using $\mathcal{O}(n^5 \log^2 C)$ bit operations in each iteration. In total, Pot-GGLLL requires

$$\mathcal{O}\left(\left(n^4 \log^2 C + mn^4 \log^2 C + n^5 \log^2 C\right) n^2 \log_{1/\delta} C\right) = \mathcal{O}\left((m+n)\,\frac{n^6 \log^3 C}{\log 1/\delta}\right)$$

bit operations. From [9][Proof of Proposition 1] we know that Pot-DeepLLL requires $\mathcal{O}\left((m+n)n^4 \log_{1/\delta} C\right)$ arithmetic operations or equivalently $\mathcal{O}\left((m+n)\frac{n^6 \log^3 C}{\log 1/\delta}\right)$ bit operations, which is the same as Pot-GGLLL. The number of bit operations for each part of the Pot-DeepLLL algorithm has been listed in Table 1.

5.2 Computational Complexity of SS-GGLLL

An upper bound on the value of SS($\mathbf{B}$) is given by

$$I_{\mathrm{SS}} = \sum_{i=1}^{n} \|\mathbf{b}_i^*\|^2 \leq n \cdot C$$

and a lower bound is $Z_{\mathrm{SS}} \geq n$ which occurs when $C = 1$. From Lemma 3, the maximum number of iterations in SS-GGLLL is

$$\log_{1/\delta}\left(\frac{I_{\mathrm{SS}}}{Z_{\mathrm{SS}}}\right) = \log_{1/\delta}(C)\,.$$

as was noted in [7, Proposition 2]. From [7, Equation 5], we know that $\Delta\mathrm{SS}_{i,k}$ can be computed in $\mathcal{O}(n^3 \log^2 C)$ bit operations. Using similar techniques as in Pot-GGLLL, the computation of $\texttt{argmax}_{1\leq i<k\leq n}(\Delta\mathrm{SS}_{i,k})$ requires $\mathcal{O}(n^4 \log^2 C)$ bit operations. Hence SS-GGLLL requires a total of

$$\mathcal{O}\left(\left(n^4 \log^2 C + mn^4 \log^2 C + n^4 \log^2 C\right) \log_{1/\delta}(C)\right) = \mathcal{O}\left(\frac{mn^4 \log^3 C}{\log 1/\delta}\right)$$

bit operations. In comparison, the number of bit operations of SS-DeepLLL is

$$\mathcal{O}\left(\frac{mn^4 \log^3 C}{\log 1/\delta}\right)$$

which is again the same as SS-GGLLL. The number of bit operations for each part of the SS-DeepLLL algorithm has been listed in Table 1.

Algorithm Name	Deep Insertion	Size Reduction	Index Search	Number of Iterations
Pot-DeepLLL	$mn^3 \log^2 C$	$mn^3 \log^2 C$	$n^4 \log^2 C$	$n^3 \log_{1/\delta} C$
Pot-GGLLL	$n^4 \log^2 C$	$mn^4 \log^2 C$	$n^5 \log^2 C$	$n^2 \log_{1/\delta} C$
SS-DeepLLL	$mn^3 \log^2 C$	$mn^3 \log^2 C$	$n^3 \log^2 C$	$n \log_{1/\delta} C$
SS-GGLLL	$n^4 \log^2 C$	$mn^4 \log^2 C$	$n^4 \log^2 C$	$\log_{1/\delta} C$

Table 1: Complexity comparison of X-DeepLLL and X-GGLLL.

Remark 8 (Comparison between X-DeepLLL and X-GGLLL)**.** *A comparison between the number of bit operations required in different parts of the* X-DeepLLL *and* X-GGLLL *algorithms is shown in Table 1. It provides a better understanding of where a greedy global algorithm makes gains and losses when compared with the corresponding* DeepLLL *algorithm. For deep insertion,* X-DeepLLL *requires* $\mathcal{O}(mn^3 \log^2 C)$ *bit operations. This involves a reordering of the basis followed by an update of the relevant GSO information. In comparison,* X-GGLLL *requires* $\mathcal{O}(n^4 \log^2 C)$ *bit operations using [11, Algorithm 4]. For size reductions,* X-DeepLLL *requires* $\mathcal{O}(n)$ *fewer bit operations than* X-GGLLL *because the greedy global algorithms need the basis to be*

completely size reduced before performing the index search. In contrast, X-DeepLLL *needs the basis to be size reduced only up to index* k *being considered in an iteration.* X-DeepLLL *also requires* $\mathcal{O}(n)$ *fewer bit operations for index search than* X-GGLLL*. In* X-DeepLLL*, the index* k *is fixed, and so only a search for the best index* i *for insertion is required. However, for* X-GGLLL*, the search covers all pairs* (i, k) *for* $1 \leq i < k \leq n$*, and so* $\mathcal{O}(n)$ *more operations are required. The increase in complexity due to index search is compensated in the number of iterations of the* `while` *loop that requires* $\mathcal{O}(n)$ *fewer operations in* X-GGLLL *than in* X-DeepLLL*. This is because* X-DeepLLL *maintains the index* k *which must reach* $k = n + 1$ *for the algorithm to terminate. If* N *is the number of deep insertions in* X-DeepLLL*, the number of times* k *is incremented in step* 13 *of Algorithm 3 is upper bounded by* $N(n-1)+n$ *as argued in [1]. In other words, there are at most* $\mathcal{O}(n)$ *more iterations of the* `while` *loop than the number of deep insertions. Since there is no such incremental change in the indices in* X-GGLLL*, hence it requires* $\mathcal{O}(n)$ *fewer iterations.*

Remark 9. *The output basis of* X-GGLLL *is* δ-X-DeepLLL *reduced just like in* X-DeepLLL*. In practice, the output basis of* X-GGLLL *is usually better than* X-DeepLLL*. However,* X-GGLLL *is not necessarily guaranteed to reduce the basis quality measure* X *more than* X-DeepLLL*.*

6 Experimental Results

We conduct a concrete comparative analysis of several relevant algorithms: LLL [1], Pot-DeepLLL [9], SS-DeepLLL [7], BKZ [6, 8] with blocksizes $\beta = 8, 10, 12$ and 20, and our two new proposals Pot-GGLLL and SS-GGLLL.

We first compare the standalone performances of the LLL-style algorithms – LLL, X-DeepLLL and X-GGLLL – with input bases that have not been preprocessed, up to dimension 150. The elements of such bases are very large integers. To ensure that the algorithms run correctly on such bases, implementations use multi-precision data types that can represent numbers using a larger number of bits than is supported by a single instruction of the underlying general-purpose processor. With increasing number of words in the precision, the time to execute an arithmetic operation on multi-precision data types increases.

In the absence of the exact floating-point precision essential for correct and most efficient execution of the multi-precision implementations of the X-DeepLLL and the X-GGLLL algorithms, we run our multi-precision implementations with overestimated precision to avoid anomalies due to floating-point arithmetic. We note from [9, Section 3], [7, Section 4] and Lemma 2 that the X-DeepLLL and the X-GGLLL algorithms terminate if and only if the basis is X-DeepLLL reduced. We utilise this fact to overestimate the precision through trial-and-error. A precision is chosen for a dimension when all algorithms successfully terminate for every input basis, producing X-DeepLLL reduced bases. As the dimension increases, these algorithms get slower due to the increase in the required precision. (We have later discussed some possible directions for improving the runtime efficiency of the X-DeepLLL and the X-GGLLL algorithms, including their multi-precision implementations.) We run all algorithms with the same precision at a given dimension. Since the BKZ algorithm starts with

a δ-LLL reduction as preprocessing before running blockwise reduction [6], it is not part of our standalone comparison.

We next compare the algorithms implemented with standard data types, running on preprocessed bases. We reduce the input bases with 0.99-LLL so that the basis entries fit into standard data types and the algorithms run much more quickly than their respective multi-precision implementations. We thus compare the performances of X-DeepLLL, X-GGLLL and BKZ with $\beta = 8, 10, 12$ and 20, on the preprocessed bases, up to dimension 210.

The reporting of our results is as follows. For every dimension, we run the algorithms on a certain number of bases and report the averages of the parameters – runtime, root Hermite factor (RHF), length of the shortest vector in the output basis, number of size reductions, and number of basis reorderings. Our first observation is that even though the output bases are *almost always* different between preprocessed (with 0.99-LLL) and standalone executions of an algorithm, they are of very similar quality. So we do not report them separately. In Section 6.1, we compare the output quality of all aforementioned LLL-style and BKZ algorithms in terms of the RHF and the length of the first vector in the reduced basis. However, the runtime behaviour of the algorithms is significantly different between standalone and preprocessed executions. Hence we report them separately in Sections 6.2 and 6.3 respectively.

For LLL, Pot-DeepLLL and Pot-GGLLL we use the threshold value $\delta = 0.99$. For SS-DeepLLL and SS-GGLLL we use the threshold $\delta = (1 - 10^{-6})$ following the rationale provided in the discussion in [7, Section 4.3.1], where they notice that in dimensions 100-150, the quality of SS-DeepLLL does not change significantly by taking δ closer to 1. For this reason, we have decided to use $\delta = (1 - 10^{-6})$ throughout our experiments.

Our Implementations

To the best of our knowledge, there is no publicly available implementation of SS-DeepLLL [7, Algorithm 2]. Hence, for the sake of uniformity and fairness, we have used our own implementations of all LLL-style algorithms in C++ using floating-point arithmetic. We used the `gcc 11.3.0` compiler and ran each algorithm on a single Intel® Xeon® Platinum 8358 CPU at 2.60 GHz on a shared memory machine.

We implement two versions of the algorithms Pot-DeepLLL, SS-DeepLLL, Pot-GGLLL and SS-GGLLL - one using standard data types `int` for integers and `long double` for rational numbers, and the other using the NTL library [34] datatypes `ZZ` for integers and `RR` for rational numbers. The NTL datatype version is used for the standalone comparisons, while the standard datatype version is used for the preprocessed comparisons. We implement LLL with the NTL datatypes for fairness in the standalone comparison.

Our implementations, the input lattice bases we use in our experiments and the outputs of our experiments are available at [12].

Input Bases

We generate 300 random bases each for dimensions $n = 40$ to $n = 210$, in steps of 10. The bases are random in the sense of Goldstein and Mayer [35] and are akin to those provided by the SVP Challenge [36]. These bases have the form

$$\mathbf{B} = \begin{bmatrix} q & \mathbf{0} \\ \mathbf{x} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} q & 0 & 0 & \dots & 0 \\ x_1 & 1 & 0 & \dots & 0 \\ x_2 & 0 & 1 & \dots & 0 \\ \vdots & & & \ddots & \\ x_{n-1} & 0 & \dots & 0 & 1 \end{bmatrix}$$

where q is a $10n$-bit prime, $\mathbf{x} = (x_1, \dots, x_{n-1})^T$ is a column vector of integers modulo q chosen uniformly at random and $\mathbf{I}$ is the $(n-1) \times (n-1)$ identity matrix.

In the standalone comparison of Section 6.2, we test all 300 bases in dimensions 40 to 90. In dimensions 100 to 150, due to the slow runtime caused by the overestimated floating-point precision, we only test the first 50 bases. In the preprocessed comparison provided in Sections 6.1 and 6.3, these bases are 0.99-LLL reduced using the FPLLL implementation [24] before being passed as input to the algorithms being compared.

Comparison with BKZ

We compare the LLL-style algorithms with BKZ only in the preprocessed phase. We use the `bkz_reduction` function of FPLLL [24] with the float type set as `long double` and the flag `BKZ_NO_LLL` since the bases are already preprocessed. All algorithms use $\eta = 0.51$ in their size reductions as is the default in the FPLLL library implementations.

Even though the runtimes of our algorithms on preprocessed bases are quite fast, we perform tests only up to dimension 210. This is because from dimension 220 onward, the FPLLL implementation of BKZ using the `long double` datatype enters an infinite loop in Babai's algorithm (i.e. the size reduction step) for some of the bases given as input. Our implementations of X-DeepLLL and X-GGLLL algorithms do not suffer from any such error, that we notice up to and including dimension 250. Hence, as far as we know, they run correctly at higher dimensions, as long as the values fit within the precision allowed by the implementation.

Experimental Data

Our comparisons of the aforementioned algorithms are based on the average values of three efficiency parameters, namely, (1) running time (Table 4 and Figure 4 for preprocessed execution; Table 6 and Figure 7 for standalone execution), (2) number of reorderings/deep insertions (Table 5 and Figure 5 for preprocessed execution; Table 7 for standalone execution), and (3) number of size reductions of the basis vectors as in step 3 of Algorithm 1 (Table 5 and Figure 6 for preprocessed execution; Table 8 for standalone execution). We measure the output quality using averages of (1) the root Hermite factor (RHF) (Table 2 and Figure 1), as well as (2) the length of the first vector in the reduced bases (Table 3 and Figure 3).

Improving Runtime

There could be several directions for improving the runtime efficiency of (multi-precision) implementations of X-DeepLLL and X-GGLLL, that may be explored in the future. Firstly, there could be smarter ways of computing the GSO information and associated bookkeeping as in [11] that has significantly improved the runtime of the X-DeepLLL algorithms, which we have already utilised. Secondly, the precision required for correct and efficient execution could be bounded as in L^2 [10] to improve the runtime of the multi-precision implementations. Moreover, one may choose the cheapest reduction strategy as in [37], which makes use of heuristics as well as methods that provably ensure that the basis is reduced not just correctly, but also efficiently. This strategy is implemented in FPLLL [24]. A similar method may be applied to the greedy global algorithms, where the overestimation of precision is used initially due to the large entries in the input basis, but the working precision is reduced as the basis quality improves.

6.1 Output Quality

We first compare the output quality of X-DeepLLL, X-GGLLL and BKZ with $\beta = 8, 10, 12, 20$. The average RHF achieved by each of these algorithms for dimensions $40 - 210$ are shown in Table 2 and Figure 1. Furthermore, the average length of the first vector in the reduced bases that give the average RHFs are shown in Table 3 and Figure 3.

	Algorithm							
Dim	Pot-Deep	Pot-GG	SS-Deep	SS-GG	BKZ-8	BKZ-10	BKZ-12	BKZ-20
40	1.01372	1.01341	1.01366	1.01333	1.01352	1.01323	1.01300	1.01243
50	1.01427	1.01390	1.01413	1.01355	1.01388	1.01344	1.01316	1.01250
60	1.01425	1.01404	1.01410	1.01373	1.01411	1.01375	1.01334	1.01241
70	1.01443	1.01418	1.01418	1.01378	1.01427	1.01382	1.01351	1.01248
80	1.01457	1.01432	1.01410	1.01378	1.01438	1.01402	1.01359	1.01250
90	1.01466	1.01453	1.01407	1.01378	1.01453	1.01404	1.01368	1.01252
100	1.01479	1.01460	1.01412	1.01379	1.01462	1.01413	1.01380	1.01260
110	1.01484	1.01471	1.01410	1.01372	1.01470	1.01416	1.01383	1.01259
120	1.01496	1.01481	1.01414	1.01375	1.01472	1.01428	1.01381	1.01267
130	1.01496	1.01484	1.01416	1.01380	1.01481	1.01425	1.01389	1.01268
140	1.01504	1.01493	1.01413	1.01377	1.01488	1.01430	1.01395	1.01267
150	1.01507	1.01501	1.01414	1.01375	1.01491	1.01434	1.01398	1.01269
160	1.01512	1.01501	1.01414	1.01375	1.01495	1.01439	1.01400	1.01269
170	1.01521	1.01508	1.01416	1.01373	1.01494	1.01442	1.01400	1.01274
180	1.01524	1.01509	1.01414	1.01375	1.01500	1.01448	1.01403	1.01273
190	1.01522	1.01511	1.01425	1.01377	1.01504	1.01447	1.01407	1.01276
200	1.01526	1.01516	1.01430	1.01383	1.01505	1.01451	1.01408	1.01276
210	1.01530	1.01519	1.01449	1.01389	1.01507	1.01451	1.01411	1.01280

Table 2: Average Root Hermite Factor (RHF) using the preprocessed input bases.

In all tested dimensions, *the average RHF achieved by* X-GGLLL *is smaller than the average RHF achieved by* X-DeepLLL. Therefore, X-GGLLL returns a better quality basis on average than the corresponding X-DeepLLL algorithm, whilst achieving

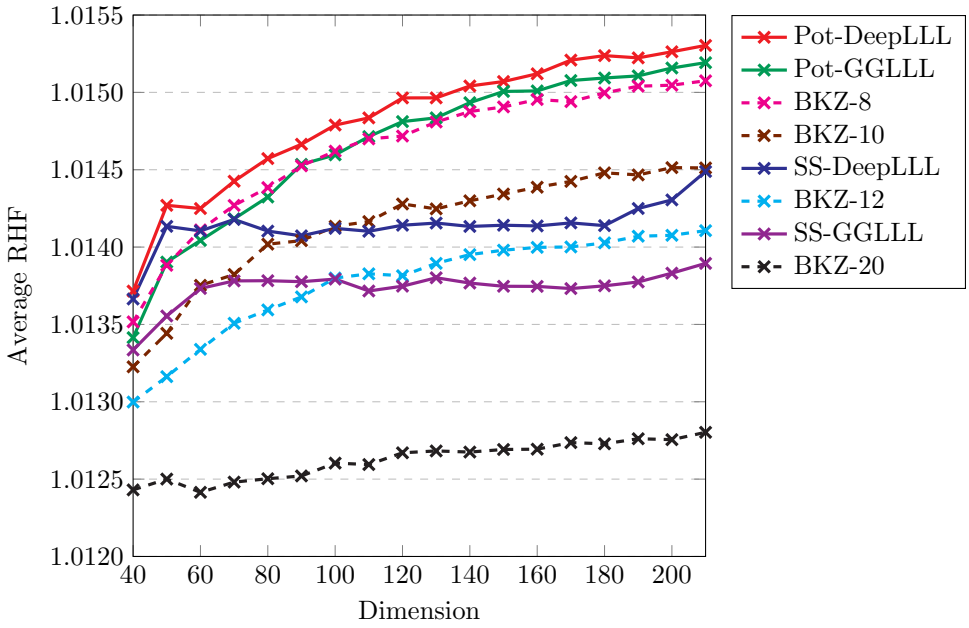

Fig. 1: The average RHF achieved by lattice basis reduction algorithms in dimensions 40 to 210 using the preprocessed input bases.

the same theoretical notion of reduction (Lemma 2) and the same asymptotic complexity (Sections 5.1, 5.2). At dimension 210, SS-GGLLL outperforms SS-DeepLLL achieving average RHFs 1.01431 (corresponding to an average $\mathbf{b}_1$ length of 18564) and 1.01516 (average $\mathbf{b}_1$ length 20999) respectively. In other words, SS-GGLLL outputs shortest vectors that are 11.6% shorter than SS-DeepLLL on the average. Similarly, Pot-GGLLL and Pot-DeepLLL achieve average RHFs (average $\mathbf{b}_1$ norms) 1.01519 (24297) and 1.01530 (24862) respectively.

When comparing with the state-of-the-art BKZ, SS-GGLLL has a smaller RHF than BKZ-8 across all tested dimensions. Furthermore, although SS-GGLLL starts below BKZ-10 and 12 in terms of its output quality at dimension 40, it eventually outperforms at higher dimensions. In Figure 2, the ratios of the average RHFs of SS-GGLLL and BKZ-$8, 10, 12$ and 20, are provided for all dimensions tested. A value above $y = 1$ implies that the RHF of SS-GGLLL is smaller, whereas a value below it implies that the RHF of BKZ is smaller. One can see from Figures 1, 2 that SS-GGLLL *starts outperforming* BKZ-10 *at around dimension* 60 *and* BKZ-12 *around dimension* 100. Furthermore, Figure 2 shows that the RHF of SS-GGLLL keeps getting better than BKZ-8, 10 and 12 as the dimension increases up to around dimension 180. From Figure 1, we also notice that the RHFs for BKZ-8, 10 and 12 are generally increasing with the dimension, while that of SS-GGLLL is reasonably consistent up until dimension 180. This explains the reason behind SS-GGLLL eventually outperforming BKZ-10 and 12 in Figure 2.

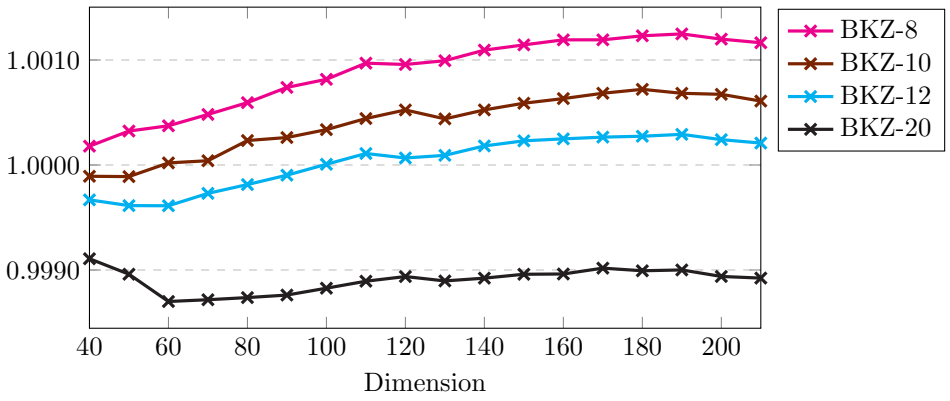

Fig. 2: The gap between the RHF of SS-GGLLL and BKZ with blocksize $\beta = 8$, 10, 12 and 20, expressed as $\frac{\text{RHF}_{\text{BKZ-}\beta}}{\text{RHF}_{\text{SS-GGLLL}}}$.

Whilst Pot-GGLLL is an improvement on Pot-DeepLLL in terms of quality, it does not compete as well with BKZ. Whilst Pot-GGLLL has an RHF less than BKZ-8 at dimension 40, their RHFs are comparable up to approximately dimension 100, BKZ-8 has a smaller RHF than Pot-GGLLL in dimension 120 and above.

We note that the RHFs achieved by lattice reduction algorithms LLL, DeepLLL and BKZ converge to a constant as the dimension increases [33]. Furthermore, the RHF achieved by these algorithms solely depends on the algorithm itself, regardless of the lattice being reduced, unless the input basis possesses a specific structure. Our greedy global algorithms also exhibit similar behaviour, and hence we can be reasonably convinced that SS-GGLLL will continue to produce bases of better quality than BKZ-12, even at higher dimensions. It is worth noting that the seemingly erratic pattern portrayed in Figure 1 is due to the small scale on the y-axis – much smaller than [33, Figure 5] – which exaggerates even the slightest distinctions in RHF. Moreover, the bases were tested only in increments of 10 in the dimension.

6.2 Standalone Runtime

We now compare the standalone runtimes of the LLL-style algorithms, without 0.99-LLL preprocessing.

Whilst the asymptotic runtime complexities (comparisons in Table 1) of our greedy global algorithms Pot-GGLLL and SS-GGLLL are the same as the corresponding X-DeepLLL algorithms, we observe in Table 6 that *our algorithms run in much less time on average in every dimension*, using overestimated precisions. Furthermore, as the dimension grows, *the greedy global algorithms become even better in comparison.* At dimension 150, SS-GGLLL is around 2.3 times faster than SS-DeepLLL and Pot-GGLLL is about 1.4 times faster than Pot-DeepLLL. In fact, SS-GGLLL is only second to LLL in standalone runtime, as is quite clear in Figure 7, being around 3.41 times slower at dimension 150.

A more granular investigation of the standalone runtime is conducted using the numbers of reorderings (deep insertions) $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ and size reductions

	Algorithm							
Dim	Pot-Deep	Pot-GG	SS-Deep	SS-GG	BKZ-8	BKZ-10	BKZ-12	BKZ-20
40	1755.2	1734.2	1751.4	1728.6	1740.6	1720.5	1705.1	1666.8
50	2070.2	2033.0	2056.3	1998.0	2030.3	1986.4	1959.0	1895.5
60	2385.1	2355.4	2363.6	2311.8	2363.7	2314.0	2257.9	2137.1
70	2783.0	2736.5	2734.7	2661.5	2751.6	2667.7	2610.1	2430.8
80	3250.9	3188.1	3131.8	3053.7	3201.7	3109.6	3007.5	2758.0
90	3790.1	3746.5	3595.2	3502.3	3740.9	3583.5	3469.7	3129.7
100	4439.5	4355.0	4154.2	4023.4	4363.5	4159.5	4023.4	3574.4
110	5168.4	5100.7	4771.7	4576.4	5089.3	4802.3	4629.6	4048.9
120	6081.1	5971.7	5516.6	5265.0	5903.2	5603.3	5302.8	4630.1
130	7058.0	6939.7	6358.7	6076.8	6915.3	6432.8	6147.6	5261.1
140	8276.4	8153.9	7299.2	6939.5	8087.2	7466.2	7114.3	5962.4
150	9650.2	9560.1	8408.9	7932.5	9415.7	8663.6	8208.7	6781.1
160	11300.2	11104.3	9671.0	9094.8	11000.7	10059.4	9457.3	7694.8
170	13330.0	13036.3	11167.5	10400.5	12739.8	11680.6	10878.8	8794.8
180	15576.5	15185.6	12813.4	11958.1	14918.2	13608.4	12559.8	9968.0
190	18076.2	17683.8	15055.7	13773.0	17458.3	15678.3	14552.4	11381.2
200	21194.1	20751.2	17538.2	15970.7	20298.2	18271.4	16756.4	12907.8
210	24861.7	24296.8	20999.1	18564.3	23707.4	21091.0	19390.3	14793.8

Table 3: Average length of the first vector in reduced bases using the preprocessed input bases.

$\mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$ performed by each algorithm. At dimension 150, the number of deep insertions of Pot-GGLLL is only 0.87% of Pot-DeepLLL, and the number of deep insertions of SS-GGLLL is only 1.87% of SS-DeepLLL, as reported in Table 7. Lesser deep insertions imply lesser iterations in X-DeepLLL and X-GGLLL. However, the number of iterations may not be the true representative of the runtime.

Remark 10. *The comparisons of the number of reorderings in* LLL*-style algorithms provide strong intuitive justification for our greedy global approach in terms of improving efficiency. One would expect that fewer reorderings of the basis (and hence fewer GSO updates and size reductions) would result in a more efficient algorithm. However, we must note that upon a reordering in the* X-GGLLL *algorithm, there is more that needs to be done compared to* X-DeepLLL *to ensure that the basis is fully size reduced for the next iteration. Hence, recording the number of size reductions of* $\mathbf{b}_k$ *with* $\mathbf{b}_j$ *(Algorithm 1[Step 3]) provides a more granular measure of efficiency for fairer comparison between the* LLL*-style algorithms.*

In the standalone comparison shown in Table 8, we see that at dimension 150 the average number of size reductions (Algorithm 1[Step 3]) of Pot-GGLLL is 61.68% of Pot-DeepLLL, and that of SS-GGLLL is 68.13% of SS-DeepLLL. These percentages are higher than the percentages of deep insertions mentioned above, because of the reasons explained in Remarks 8 and 10. However, the decrease in the number of deep insertions and iterations is so prominent, that the additional operations after every deep insertion is well compensated. In summary, X-GGLLL *requires significantly fewer deep insertions and overall fewer size reductions in standalone comparison than* X-DeepLLL. Hence the overall runtime of the X-GGLLL algorithms is much better than the X-DeepLLL algorithms.

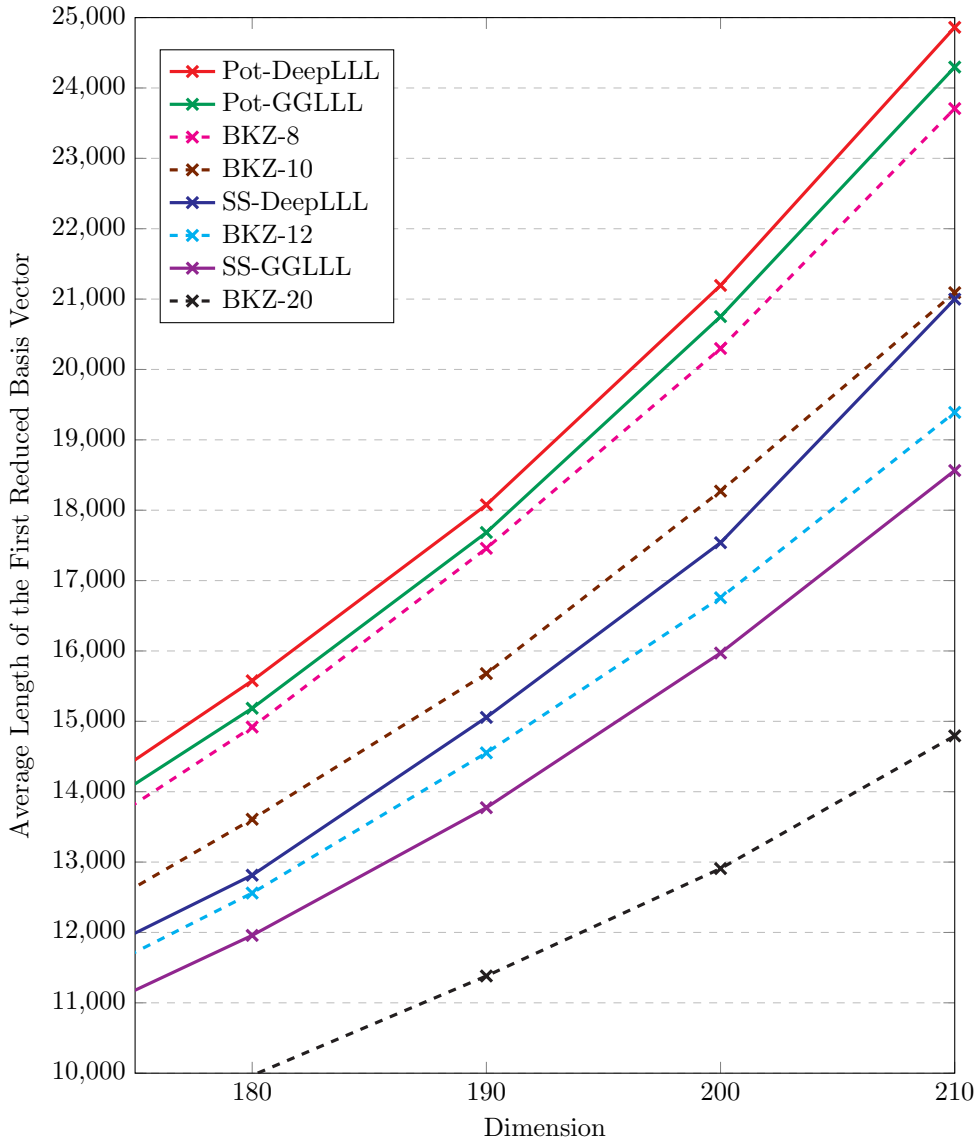

Fig. 3: The average length of the first basis vector after being reduced in dimensions 180 to 210 using the preprocessed input bases.

We reiterate that the overestimated floating-point precision is key to the runtimes of the X-DeepLLL and X-GGLLL algorithms. It may be possible to improve the runtimes of these algorithms using techniques from [10, 37, 11]. However, such improvements should not change the number of deep insertions or size reductions reported here.

	Algorithm							
Dim	Pot-Deep	Pot-GG	SS-Deep	SS-GG	BKZ-8	BKZ-10	BKZ-12	BKZ-20
40	0.00146	0.00287	0.00134	0.00165	0.0276	0.0301	0.0336	0.0582
50	0.00753	0.00377	0.00316	0.00386	0.0743	0.0835	0.0931	0.166
60	0.00953	0.0185	0.00783	0.00915	0.165	0.184	0.214	0.426
70	0.0220	0.0409	0.0172	0.0209	0.347	0.381	0.427	0.927
80	0.0439	0.0861	0.0352	0.0438	0.684	0.733	0.828	1.96
90	0.0800	0.155	0.0632	0.0824	1.23	1.33	1.48	3.62
100	0.141	0.286	0.111	0.155	2.01	2.21	2.48	6.80
110	0.232	0.463	0.184	0.276	3.02	3.42	3.90	11.6
120	0.364	0.742	0.287	0.468	4.48	5.33	5.83	17.7
130	0.545	1.18	0.435	0.775	6.72	7.36	8.32	26.5
140	0.814	1.75	0.651	1.23	9.82	10.6	11.5	39.2
150	1.17	2.55	0.94	1.94	13.4	15.3	16.9	54.6
160	1.65	3.68	1.34	3.01	20.7	24.3	27.1	88.2
170	2.26	5.23	1.85	4.44	27.3	30.0	34.3	125
180	3.04	7.10	2.46	6.37	34.8	38.5	42.6	172
190	4.00	9.79	3.18	8.99	46.1	49.7	55.6	226
200	5.36	13.2	4.16	12.6	58.8	64.1	72.3	285
210	6.73	16.8	4.84	16.5	74.7	85.2	97.8	364

Table 4: Average runtime in seconds (rounded to most significant 3 digits) on the preprocessed bases.

6.3 Runtime with LLL Preprocessing

Here we compare the runtime of the algorithms on SVP Challenge bases which have been 0.99-LLL reduced using the FPLLL [24] implementation.

Our preprocessed runtime results are in contrast with the standalone runtime comparison from Section 6.2. We see from Table 4 (Figure 4) that *our algorithms have a slower runtime than the corresponding* DeepLLL *algorithm* on the 0.99-LLL reduced bases used as input. Furthermore, as the dimension grows, the gap between X-DeepLLL and X-GGLLL slowly increases. At dimension 40, Pot-GGLLL is 2.0× slower than Pot-DeepLLL and SS-GGLLL is 1.2× slower than SS-DeepLLL. At dimension 210, the greedy global algorithms are 2.5× and 3.4× slower respectively. However, the greedy global algorithms are still very efficient in practice; reducing the input bases at dimension 210 in less than 17 seconds on average.

As before, we consider the subroutines within each iteration to conduct a granular analysis of the runtime differences between X-DeepLLL and X-GGLLL. We consider the numbers of basis reorderings (deep insertions) $\mathbf{B} \leftarrow \sigma_{i,k}(\mathbf{B})$ and size reductions $\mathbf{b}_k - \lfloor \mu_{k,j} \rceil \, \mathbf{b}_j$ (Algorithm 1[Step 3]) performed by each of the algorithms. The average number of reorderings in the LLL-style algorithms is provided in Table 5 (Figure 5). As in the standalone comparison, across all tested dimensions, *the greedy global algorithms perform fewer deep insertions than their* DeepLLL *counterparts*. At dimension 40, Pot-DeepLLL and SS-DeepLLL perform 3.0× more deep insertions than Pot-GGLLL and SS-GGLLL respectively. At dimension 210, Pot-DeepLLL performs 17.6× more deep insertions than Pot-GGLLL and SS-DeepLLL performs 5.5× more deep insertions than SS-GGLLL. It is also interesting to note here that Pot-GGLLL performed *fewer reorderings* than SS-GGLLL on average across all tested dimensions.

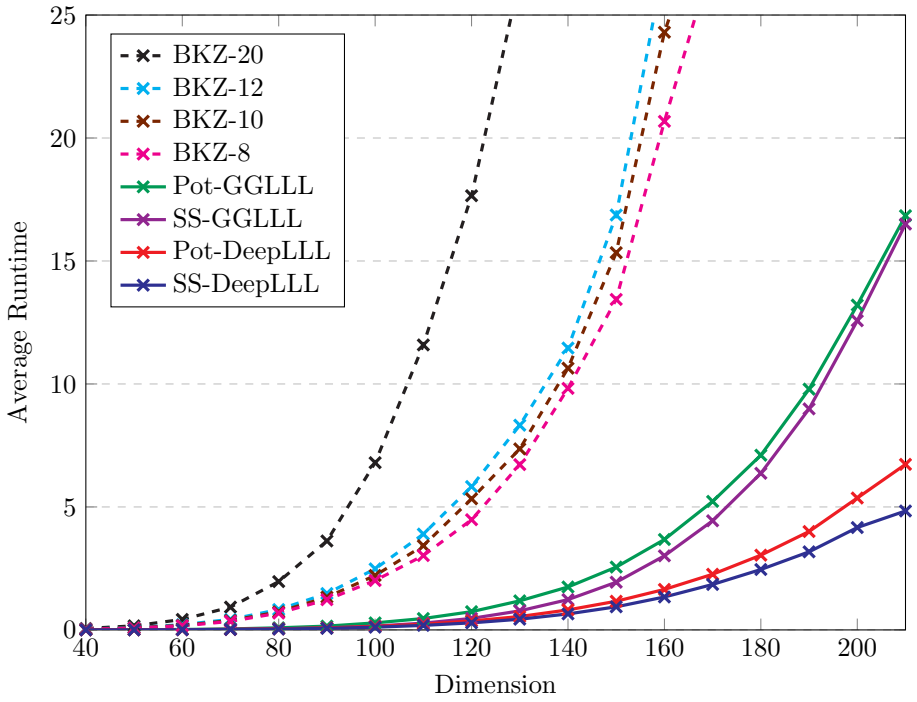

Fig. 4: The average runtime of lattice basis reduction algorithms in dimensions 40 to 210 using the preprocessed input bases.

As pointed out in Remark 10, the average number of size reductions (Algorithm 1[Step 3]) is a more granular indicator for the runtime. Table 5 also provides the average number of size reductions. Across all tested dimensions *the greedy global variants always perform more size reductions than their* DeepLLL *counterparts.* At dimension 40, Pot-GGLLL and SS-GGLLL perform 1.9× more size reductions than Pot-DeepLLL and SS-DeepLLL respectively. At dimension 210, Pot-GGLLL performs 2.4× more size reductions than Pot-DeepLLL and SS-GGLLL performs 3.9× more size reductions than SS-DeepLLL. Furthermore, a graphical representation of the average number of size reductions in Figure 6 shows the diverging curves of X-GGLLL and the respective X-DeepLLL counterparts showing that *the difference in the number keeps growing as the dimension increases.* Unlike the standalone performance of the algorithms, the reduction in the number of reorderings in the preprocessed performance of the greedy global framework has not been able to compensate for the increase in its number of size reductions. This, along with the index search step, appears to be the reason why X-GGLLL is slower than X-DeepLLL in practice using standard data type implementations for preprocessed bases.

When comparing with BKZ, we see that *both* SS-GGLLL *and* Pot-GGLLL *are unilaterally faster than the very efficient FPLLL [24] implementation of* BKZ *using the* `long double` *datatype for GSO information for all* 4 *blocksizes and all dimensions in*

	Algorithm							
	Reorderings				Size Reductions			
Dim	Pot-Deep	Pot-GG	SS-Deep	SS-GG	Pot-Deep	Pot-GG	SS-Deep	SS-GG
40	211	71	247	83	2620	4927	3224	5987
50	502	122	584	143	8470	15524	10357	18917
60	983	180	1170	216	21509	37996	27524	47166
70	1716	253	2045	315	46684	81529	60196	104540
80	2740	358	3359	450	90205	161833	122865	213469
90	4058	457	4966	604	158490	284433	220329	392239
100	5881	604	7151	826	264844	490594	377831	705232
110	8042	733	9739	1110	413640	770525	610176	1200356
120	10748	907	12811	1464	624169	1190832	938518	1960126
130	13820	1138	16178	1918	893756	1792218	1384967	3104523
140	17938	1338	20411	2440	1282404	2584909	2026213	4742813
150	22354	1594	24644	3162	1754879	3619918	2845232	7133488
160	27284	1840	29193	3986	2331875	4931985	3882114	10412952
170	33425	2195	34349	4925	3086531	6751534	5235544	14816911
180	40278	2516	39459	5988	4005652	8927973	6840290	20568885
190	47565	2935	44186	7163	5058505	11673474	8599039	27791502
200	56683	3363	49229	8413	6438373	15158279	10614901	36908973
210	66134	3766	53005	9715	7998874	19132055	12410508	47824142

Table 5: Average number of reorderings and size reductions $\mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$ for the preprocessed bases.

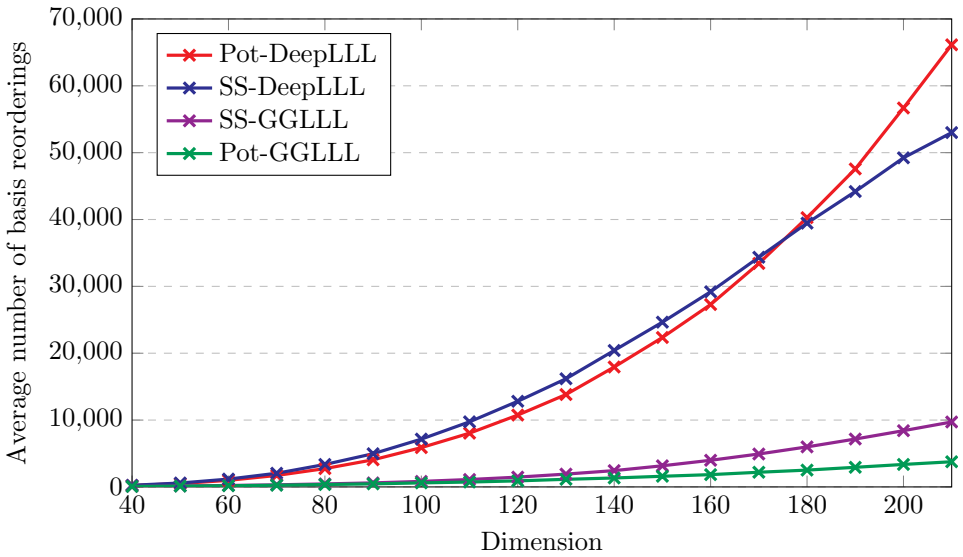

Fig. 5: The average number of basis reorderings required to reduce bases in dimensions 40-210.

our tests. As a result, since X-DeepLLL is faster than X-GGLLL on the preprocessed bases, we also have that Pot-DeepLLL and SS-DeepLLL are also unilaterally faster than BKZ. At dimension 150, BKZ-8, 10, 12 and 20 took around 6.9, 7.9, 8.7 and

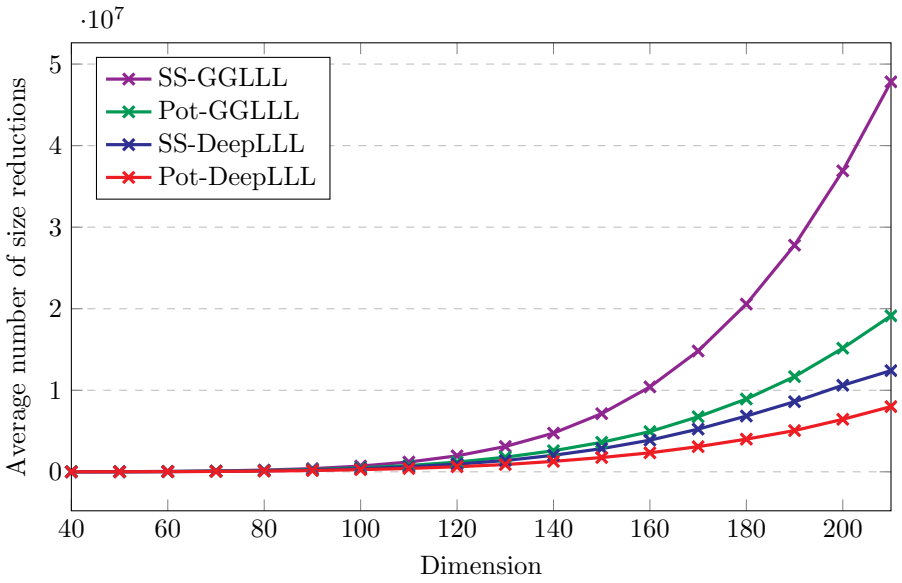

Fig. 6: The average number of individual size reductions $\mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$ to reduce bases in dimensions 40-210 using the preprocessed input bases.

28.1 times longer than SS-GGLLL respectively. At dimension 210, BKZ-8, 10, 12 and 20 took around 4.5, 5.2, 5.9 and 22.1 times longer than SS-GGLLL respectively . It is clear from the diverging curves in Figure 4 that BKZ *becomes slower compared to* SS-GGLLL *as the dimension increases.*

7 Conclusion

In this work, we first interpreted the (generalised) Lovász condition [6] as a reordering constraint used to improve the length $\|\mathbf{b}_i^*\|$ of the i^{th} GSO vector of the basis, which is a localised measure of the quality of the basis. We thus arrived at a generalised representation of DeepLLL [6], Pot-DeepLLL [9] and SS-DeepLLL [7] algorithms where they iteratively improve a local or global measure of basis quality. This generalisation leads us to the new greedy global framework in the form of a generic algorithm X-GGLLL for lattice basis reduction. The algorithm works by iteratively reducing a general measure of quality X of the basis. The key novelty in the framework is to work with the whole lattice in every iteration in place of a sublattice as in all previous LLL-style algorithms. The basis vectors are reordered using a greedy approach towards improving their quality X which results in very efficient algorithms. Our framework is instantiated by substituting the general measure X with the concrete quality measures potential (Pot) and squared sum (SS). A local measure like $\mathrm{LC}_i(\mathbf{B})$ provides independent values for different indices i. So more careful consideration must be taken to construct a greedy global algorithm using such a measure, that we leave as future work. We have proved results on the efficiency of the generic algorithm X-GGLLL

and on the two new concrete algorithms Pot-GGLLL and SS-GGLLL. Furthermore, we have shown that the bases produced by our algorithms are of provable quality.

Using multi-precision arithmetic implementations for standalone comparison between the algorithms (without preprocessing the bases), our greedy global variants have a faster runtime than their DeepLLL counterparts, whilst also outperforming them in terms of basis quality. In fact, SS-GGLLL is only second to LLL in standalone runtime, while of course providing much shorter vectors. However, using standard data type implementation for 0.99-LLL preprocessed input bases, the X-GGLLL algorithms are slower than the corresponding X-DeepLLL algorithms, whilst still outperforming them in quality. We provide justifications for the runtime comparisons based on more granular runtime parameters like the number of reorderings pertaining to iterations and the number of size reductions. Our implementations are public and it may be possible to improve the runtimes of our X-GGLLL algorithms using techniques similar to those [10, 37, 11] that have already been used on previous LLL-style algorithms.

Note that while working on the preprocessed bases, the X-DeepLLL and X-GGLLL algorithms are significantly quicker than the FPLLL implementation of BKZ, even for small blocksizes. Furthermore, SS-GGLLL outputs better quality bases than BKZ-12 in dimension 100 and greater. In other words, SS-GGLLL is an improvement on BKZ-12 in both runtime efficiency and basis quality. This demonstrates the practical strength of the greedy global approach.

Our design principle has been to achieve the best possible efficiency in reaching an assured quality by reducing the measure X as much as possible in each iteration. The result is quick improvements in the basis quality. Our framework could be altered to not make the most greedy choice resulting in a slower algorithm which performs more iterations to help close the gap in quality with BKZ (say with blocksize 20) at smaller dimensions. There could be other strategies that may as well decrease the overall runtime without compromising on the quality or even improving it, as X-GGLLL did over X-DeepLLL. One may also consider employing more than one basis quality measure and their combinations to be improved. We leave such explorations for future work with the belief that our framework has opened up avenues for designing interesting new lattice reduction algorithms.

Acknowledgements.

We thank the anonymous reviewers for their detailed comments on an earlier version of this paper that helped in improving it significantly. We also thank Palash Sarkar for his helpful comments on the paper.

References

[1] Arjen K. Lenstra, Hendrik Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

[2] Phong Q. Nguyen and Jacques Stern. The two faces of lattices in cryptology. In Joseph H. Silverman, editor, *Cryptography and Lattices*, pages 146–180, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[3] Henri Cohen. *A course in computational algebraic number theory.* Springer, 1993.

[4] Jürgen Klüners. *The van Hoeij algorithm for factoring polynomials*, pages 283–291. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
[5] Guillaume Hanrot. *LLL: A tool for effective Diophantine approximation*, pages 215–263. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
[6] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66(1):181–199, 1994.
[7] Masaya Yasuda and Junpei Yamaguchi. A new polynomial-time variant of LLL with deep insertions for decreasing the squared-sum of Gram–Schmidt lengths. *Designs, Codes and Cryptography*, 87(11):2489–2505, 2019.
[8] Yuanmi Chen and Phong Q. Nguyen. BKZ2.0: Better lattice security estimates. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 1–20, Seoul, South Korea, December 4–8, 2011. Springer.
[9] Felix Fontein, Michael Schneider, and Urs Wagner. PotLLL: a polynomial time version of LLL with deep insertions. *Designs, Codes and Cryptography*, 73(2):355–368, 2014.
[10] Phong Q. Nguyen and Damien Stehlé. An LLL algorithm with quadratic complexity. *SIAM Journal on Computing*, 39(3):874–903, 2009.
[11] Junpei Yamaguchi and Masaya Yasuda. Explicit formula for gram-schmidt vectors in LLL with deep insertions and its applications. In Jerzy Kaczorowski, Josef Pieprzyk, and Jacek Pomykała, editors, *Number-Theoretic Methods in Cryptology*, pages 142–160, Cham, 2018. Springer International Publishing.
[12] Sanjay Bhattacherjee and Jack Moyler. Our implementations of some LLL-style algorithms, 2023. https://github.com/GG-LLL/Greedy-Global-LLL.
[13] Masaya Yasuda, Kazuhiro Yokoyama, Takeshi Shimoyama, Jun Kogure, and Takeshi Koshiba. Analysis of decreasing squared-sum of gram-schmidt lengths for short lattice vectors. *Journal of Mathematical Cryptology*, 11(1):1–24, 2017.
[14] Masaharu Fukase and Kenji Kashiwabara. An accelerated algorithm for solving SVP based on statistical analysis. *J. Inf. Process.*, 23:67–80, 2015.
[15] Ali Akhavi. The optimal LLL algorithm is still polynomial in fixed dimension. *Theoretical Computer Science*, 297(1):3–23, 2003.
[16] Nicholas A. Howgrave-Graham. Isodual reduction of lattices. Cryptology ePrint Archive, Paper 2007/105, 2007. https://eprint.iacr.org/2007/105.
[17] Jingwei Chen, Damien Stehlé, and Gilles Villard. Computing an LLL-reduced basis of the orthogonal lattice. In Carlos Arreche, editor, *Proceedings of the 2018 ACM International Symposium on Symbolic and Algebraic Computation*, ISSAC '18, page 127–133, New York, NY, USA, 2018. Association for Computing Machinery.
[18] Ivan Morel, Damien Stehlé, and Gilles Villard. H-LLL: using householder inside LLL. In John P. May, editor, *Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation*, ISSAC '09, page 271–278, New York, NY, USA, 2009. Association for Computing Machinery.
[19] Xiao-Wen Chang, Damien Stehlé, and Gilles Villard. Perturbation analysis of the QR factor R in the context of LLL lattice basis reduction. *Mathematics of Computation*, 81(279):1487–1511, 2012.

[20] Thomas Debris-Alazard, Léo Ducas, and Wessel P. J. van Woerden. An algorithmic reduction theory for binary codes: LLL and more. *IEEE Transactions on Information Theory*, 68(5):3426–3444, 2022.

[21] Hendrik W. Lenstra. Flags and lattice basis reduction. In Carles Casacuberta, Rosa Maria Miró-Roig, Joan Verdera, and Sebastià Xambó-Descamps, editors, *European Congress of Mathematics*, pages 37–51, Basel, 2001. Birkhäuser Basel.

[22] Paul Kirchner, Thomas Espitau, and Pierre-Alain Fouque. Towards faster polynomial-time lattice reduction. In Tal Malkin and Chris Peikert, editors, *2021, Part II*, volume 12826 of *LNCS*, pages 760–790, Virtual Event, August 16–20, 2021. Springer.

[23] Keegan Ryan and Nadia Heninger. Fast practical lattice reduction through iterated compression. Cryptology ePrint Archive, Paper 2023/237, 2023. https://eprint.iacr.org/2023/237.

[24] The FPLLL development team. FPLLL, a lattice reduction library, Version: 5.4.4. Available at https://github.com/fplll/fplll, 2023.

[25] Henrik Koy and Claus Peter Schnorr. Segment LLL-reduction of lattice bases. In Joseph H. Silverman, editor, *Cryptography and Lattices*, pages 67–80. Springer Berlin Heidelberg, 2001.

[26] Arnold Neumaier and Damien Stehlé. Faster LLL-type reduction of lattice bases. In Markus Rosenkranz, editor, *Proceedings of the ACM on International Symposium on Symbolic and Algebraic Computation*, ISSAC '16, page 373–380, New York, NY, USA, 2016. Association for Computing Machinery.

[27] Thomas Plantard, Willy Susilo, and Zhenfei Zhang. LLL for ideal lattices: re-evaluation of the security of gentry—halevi's fhe scheme. *Des. Codes Cryptography*, 76(2):325–344, aug 2015.

[28] Changmin Lee, Alice Pellet-Mary, Damien Stehlé, and Alexandre Wallet. An LLL algorithm for module lattices. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 59–90, Kobe, Japan, December 8–12, 2019. Springer.

[29] Tristram Bogart, John Goodrick, and Kevin Woods. A parametric version of LLL and some consequences: Parametric shortest and closest vector problems. *SIAM J. Discret. Math.*, 34(4):2363–2387, jan 2020.

[30] Phong Q. Nguyen and Damien Stehlé. LLL on the average. In Florian Hess, Sebastian Pauli, and Michael Pohst, editors, *Proceedings of the 7th International Conference on Algorithmic Number Theory*, ANTS'06, page 238–256, Berlin, Heidelberg, 2006. Springer-Verlag.

[31] Michael Schneider, Johannes Buchmann, and Richard Lindner. Probabilistic analysis of LLL reduced bases. In Johannes A. Buchmann, John Cremona, and Michael E. Pohst, editors, *Algorithms and Number Theory*, volume 9221 of *Dagstuhl Seminar Proceedings (DagSemProc)*, pages 1–6, Dagstuhl, Germany, 2009. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[32] Steven D. Galbraith. *Mathematics of public key cryptography*. Cambridge University Press, USA, 1st edition, 2012.

[33] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 31–51, Istanbul,

Turkey, apr 2008. Springer.
[34] Victor Shoup. NTL: A library for doing number theory. Available at https://github.com/libntl/ntl, 2021.
[35] Daniel Goldstein and Andrew Mayer. On the equidistribution of hecke points. *Forum Mathematicum*, 15(2):165–189, 2003.
[36] Darmstadt T.U. SVP challenge. https://www.latticechallenge.org/svp-challenge/.
[37] Damien Stehlé. *Floating-point LLL: theoretical and practical aspects*, pages 179–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

A Standalone Comparison Data

	Algorithm				
Dim	LLL	Pot-Deep	Pot-GG	SS-Deep	SS-GG
40	2.02	14.2	6.37	8.54	2.19
50	5.74	48.1	20.7	25.3	6.06
60	12.6	136	59.1	63.2	15.1
70	22.7	295	134	124	31.1
80	41.9	645	301	244	63.2
90	85.0	1323	649	490	120
100	148	3022	1555	954	275
110	248	5644	3043	1777	497
120	348	8918	5130	2479	816
130	580	15929	9865	4103	1695
140	831	25617	16886	6268	2554
150	1134	37328	26738	8752	3867

Table 6: Average runtime in seconds (rounded to most significant 3 digits for smaller values) for the standalone algorithms.

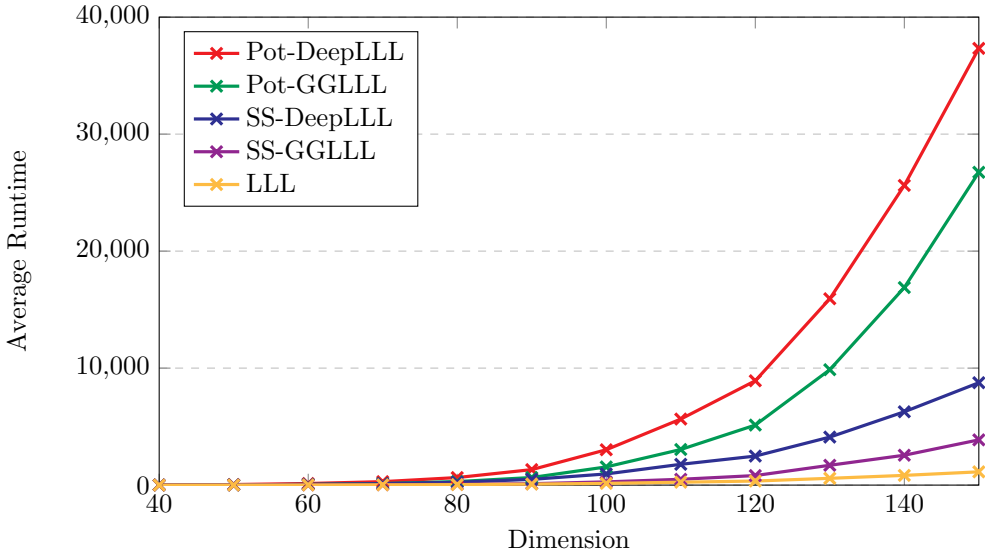

Fig. 7: The average runtime of the standalone algorithms in dimensions 40 to 150.

	Algorithm				
Dim	LLL	Pot-Deep	Pot-GG	SS-Deep	SS-GG
40	27265	11891	301	12512	362
50	47667	22251	429	23996	517
60	73652	36332	571	39682	705
70	105302	54153	737	59628	927
80	142102	75783	914	83783	1206
90	184177	101189	1128	112048	1525
100	231595	130547	1364	144138	1938
110	282967	163159	1578	179535	2400
120	339909	199762	1887	217762	2930
130	400127	239663	2186	258387	3626
140	465034	283244	2548	301361	4429
150	535176	329958	2902	346082	5442

Table 7: Average number of basis reorderings (rounded to nearest whole number) for the standalone algorithms.

	Algorithm				
Dim	LLL	Pot-Deep	Pot-GG	SS-Deep	SS-GG
40	80917	127230	67410	143755	70706
50	185909	320840	155502	371094	164548
60	359996	669874	312213	790779	337483
70	623946	1228812	570384	1480132	630993
80	992473	2052506	963137	2522662	1106309
90	1485987	3199109	1541376	4010701	1827259
100	2123061	4732812	2353368	6044032	2903142
110	2901939	6677443	3417334	8694534	4428745
120	3859389	9122406	4894355	12063243	6553179
130	4982003	12083865	6759215	16242772	9523874
140	6281915	15604490	9173358	21403549	13473035
150	7815673	19710300	12158151	27623812	18822813

Table 8: Average number of size reductions $\mathbf{b}_k - \lfloor \mu_{k,j} \rceil \mathbf{b}_j$ (rounded to nearest whole number) for the standalone algorithms.