Version: April 17, 2025

Lattice Tester Guide

C++ software tools for measuring the uniformity of integral lattices in the real space

Pierre L'Ecuyer

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal

Abstract: Lattice Tester is a C++ software library to compute theoretical measures of uniformity (or figures of merit) for lattices in the t-dimensional integer space $\mathbb{Z}^t$ (all points have integer coordinates). Such lattices are encountered for example (after rescaling) in quasi-Monte Carlo integration by lattice rules and in the analysis of uniform random number generators defined by linear recurrences modulo a large integer. Measures of uniformity include the length of the shortest nonzero vector in the lattice or in its dual (the spectral test), the Beyer ratio, as well as figures of merit that take normalized versions of these measures over projections of the lattice on subsets of the t coordinates, and then take a weighted sum or the worst-case over the class of considered projections. Lattice Tester is used in particular in the LatNet Builder and LatMRG software tools, designed to construct and analyze lattice rules and multiple recursive linear congruential generators, respectively.

Keywords: integration lattice; lattice structure; shortest vector; spectral test; LLL reduction; BKZ reduction.

Acknowledgement: François Blouin, Erwan Bourceret, Anna Bragina, Ajmal Chaumun, Raymond Couture, Marco Jacques, François Paradis, Marc-Antoine Savard, Richard Simard, Mamadou Thiongane, Josée Turgeon, and Christian Weiß have contributed to this software.

1 Introduction

The purpose of Lattice Tester is to compute certain *figures of merit* (FOMs) that serve as measures of uniformity for *integral lattices* in the t-dimensional space $\mathbb{Z}^t$, i.e., for which all the lattice points have integer coordinates. For general background on lattices in the real space, see for example [5, 8, 70, 81]. The types of lattices that we consider here are lattices L_t in the real space $\mathbb{R}^t$ that contain $\mathbb{Z}^t$ and whose points have coordinates that are all integer multiples of $1/m$ for some integer $m > 0$. The integrality property is obtained by rescaling the original lattice L_t by the integer factor m, so we work with the rescaled lattice $\Lambda_t = mL_t$ instead of the original one.

The lattices L_t are discrete vector spaces in $\mathbb{R}^t$ and one is often interested in the (finite) intersection of L_t with the t-dimensional unit hypercube $[0,1)^t$. This is the case for *lattice rules*, which are multivariate integration methods that take the average value of a function at this set of points (sometimes randomly shifted modulo 1) to estimate the integral of the function over the unit cube [36, 38, 50, 52, 81]. Bounds on the worst-case or mean square integration error, for example, can be obtained in terms of FOMs that measure the uniformity of the lattice. This same set of points $L_t \cap [0,1)^t$ is also the set of all vectors of t successive values produced by certain types of linear *random number generators (RNGs)* in scalar or matrix form. This includes *linear congruential generators (LCGs)*, matrix LCGs, (linear) *mutiple recursive generators (MRGs)*, an combinations of these [9, 30, 58, 44, 31, 35, 57, 61]. In these settings, we want the point set to cover the unit hypercube as uniformly as possible.

A standard way of measuring this uniformity is the *spectral test* [11, 25, 47, 34], that computes the length of a shortest nonzero vector in the dual lattice. The inverse of this length represents the maximal distance between successive hyperplanes in a family of equidistant hyperplanes that contain all the lattice points. We want this maximal distance to be not too large, for the points to cover the space evenly. Lattice Tester permits one to compute this distance for the lattice and for any of its projections over a subset of coordinates, which are also lattices. This is the typical target task that the software has been designed for. We compute a shortest nonzero lattice vector using a *branch-and-bound* (BB) integer programming optimization procedure which is a slightly modified version of the algorithm of [14].

An important difference between lattice rules and the lattices obtained from RNGs, from the practical viewpoint, is that the number η of lattice points per unit of volume, i.e., the number of lattice points in the unit hypercube $[0,1)^t$, called the lattice density, is usually modest (at most in the millions) for lattice rules, but much larger (say, 2^{200} or more) for RNGs. For this reason, the FOMs used for these two applications typically differ. The most popular measures for lattice rules (e.g., $\mathcal{P}_\alpha$ with weights) can be computed in a time that increases at least linearly with η and are therefore unusable when $\eta > 2^{100}$. The spectral test, whose computing time is exponential in t in the worst case but only logarithmic in η, is more appropriate for such large values of η, and therefore to test the lattice structure of RNGs. It has also been used to measure the uniformity of lattice rules [20, 50, 62, 63].

In this document, we denote the lattice dimension by t, following many papers and books on LCGs; e.g., [25, 30, 31]. Most papers on lattice rules use s or d instead, while most books and papers on lattice reduction theory use n. We define the dimension t as the number of coordinates in the lattice vectors. It could happen in general that the lattice has a smaller dimension in the sense that it contains less than t independent vectors. But this does not occur in our setting.

The rest of the document is organized as follows. In Section 2, we give a brief historical account on how this software grew. In Section 3, we define the lattices considered here and recall their main properties. Section 4 lists the main problems addressed by the software. Section 5 explains how to obtain a basis (triangular or not) from a set of (possibly dependent) generating vectors. Section 6 shows how to compute the m-dual of a given basis. In Section 7, we review different forms of lattice basis reduction: Korkine-Zolotarev and Minkowski-reduced basis, pairwise reduction, LLL reduction with factor δ, and blockwise Korkine-Zolotarev reduction with factor δ. Applying some of these reductions before finding a shortest vector with the BB algorithm is important (often essential) for performance. Section 8 outlines our BB procedure to compute a shortest nonzero vector in the lattice. The main version is for the L^2 norm, but it also works for the L^1 norm, which is computationally more expensive. In Section 9 and 10, we define and discuss the various normalizations and figures of merit that the software can compute. In Section 11, we summarize what the software is doing and how it is organized. Section 12 describes several examples of programs that use the library and illustrate key properties of the algorithms, including their performance.

Lattice Tester is implemented as a C++ library meant to be used by other software. It is currently used by LatMRG [46] and by LatNet Builder [49]. The software is available on GitHub [45].

2 Historical notes

The first version of this software was a single Pascal program written around 1985, which implemented the spectral test algorithm given in [13, 25]. It was used for [27, 28, 29]. An new version written in the Modula-2 language in 1988 incorporated procedures to search for good multiple recursive generators based on figures of merit that involved normalized version of the spectral test [42]. It used SENTIERS [55] for the arithmetic with arbitrarily large integers. That version was used for [43, 44]. Several improvements led to the Modula-2 version of the LatMRG [47, 48], which was used for [31, 41, 33, 34, 56, 59, 60], for example. LatMRG incorporated the better algorithm from [14] to compute a shortest nonzero vector in a lattice, the LLL algorithm for pre-reduction, computing a Minkowski-reduced basis, various types of representation of numbers, various figures of merit, searching procedures, and more.

Around 2000, Modula-2 compilers were no longer available and we decided to port LatMRG to C++. Between 2000 and 2010, Richard Simard made a rough translation from Modula-2 (which was not an object-oriented language) to C++. Several students also made

various changes and additions along the years. For the arithmetic with large numbers, we switched from SENTIERS to GMP and NTL. In 2014, David Munger separated LatMRG in two pieces: (1) Latcommon, which became Lattice Tester in 2017) [53], whose task is to compute measures of uniformity for arbitrary integer lattices, and (2) the new LatMRG which uses those facilities to test the lattice structure of linear RNGs and to search for good generators under various types of constraints. The rationale for having Lattice Tester (or Latcommon) as a separate software was that it can be used to analyze lattices for other purposes than testing or selecting RNGs. Latcommon was also used in 2014 in Lattice Builder[53] and Lattice Tester is now also used in LatNet Builder [51].

3 Lattices in the real space and in the integer space

3.1 Integral lattices in the real space

We consider *lattices* over $\mathbb{Z}$ (or $\mathbb{Z}$-lattices) in the real space $\mathbb{R}^t$, which are discrete subspaces of the real vector space $\mathbb{R}^t$ that can be expressed as

$$L_t = \sum_{j=1}^{t} \mathbb{Z}\boldsymbol{v}_j = \left\{ \boldsymbol{v} = \sum_{j=1}^{t} z_j \boldsymbol{v}_j \mid \text{ each } z_j \in \mathbb{Z} \right\}, \tag{1}$$

where t is a positive integer, and $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ are linearly independent (nonzero) vectors in $\mathbb{R}^t$ which form a *basis* of the lattice. The matrix $\boldsymbol{V}$, whose ith *row* is $\boldsymbol{v}_i$, is a *generator matrix* of L_t. In this document, unless indicated otherwise, the vectors are row vectors and their indices start at 1, to agree with the usual mathematical notation. (In the software code, they start at 0.) We can write $\boldsymbol{v} = (v_1, \ldots, v_t)$ in (1) as $\boldsymbol{v} = \boldsymbol{z}\boldsymbol{V}$ where $\boldsymbol{z} = (z_1, \ldots, z_t)$. If we multiply $\boldsymbol{V}$ by any unimodular $t \times t$ matrix, we get another basis, and any basis can be obtained in this way [75].

The determinant of the matrix $\boldsymbol{V}$ is equal to the volume of the fundamental parallelepiped defined as $\{\boldsymbol{v} = \lambda_1 \boldsymbol{v}_1 + \cdots + \lambda_t \boldsymbol{v}_t \mid 0 \le \lambda_i \le 1 \text{ for } 1 \le i \le t\}$. It is independent of the choice of basis. It is called the *fundamental volume* (or determinant) of L_t, and denoted $\det(L_t) = \det(\boldsymbol{V})$. The quantity $\eta_t = 1/\det(L_t) = 1/\det(\boldsymbol{V}) = \det(\boldsymbol{V}^{-1})$ is called the *density* of L_t and it represents the average number of points per unit of volume. When L_t contains $\mathbb{Z}^t$, the density η_t is an integer equal to the cardinality of the point set $L_t \cap [0,1)^t$. In our setting, this will always occur.

For a given lattice L_t and a subset of coordinates $I = \{i_1, \ldots, i_s\} \subseteq \{1, \ldots, t\}$, we denote by L_I the *projection* of L_t over the s-dimensional subspace determined by the coordinates in I. This projection is also a lattice, whose density η_I divides that of L_t. This density η_I is often equal to η_t, but is sometimes smaller. There are exactly $\det(L_I)/\det(L_t) = \eta_t/\eta_I$ points of L_t that are projected onto each point of L_I. In group theory language, L_I corresponds to a coset of L_t.

A *shifted lattice* is a lattice L_t shifted by a constant vector $\boldsymbol{v}_0 \notin L_t$, i.e., a point set of the form $L'_t = \{\boldsymbol{v} + \boldsymbol{v}_0 : \boldsymbol{v} \in L_t\}$, where L_t is a lattice. The uniformity of a shifted lattices L'_t can be analyzed by subtracting the shift and analyzing the (unshifted) lattice L_t.

We assume that a norm $\|\cdot\|$ in $\mathbb{R}^t$ has been selected to measure the length of lattice vectors. By default, it is the L^2 (Euclidean) norm, but in some situations we may use a different one, for example the L^1 norm. We denote the norm of $\boldsymbol{v}$ by $\|\boldsymbol{v}\|$. We also denote by $\boldsymbol{v} \cdot \boldsymbol{w}$ the *standard* scalar product of vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ (the inner product that corresponds to the L^2 norm).

The *dual lattice* of L_t is defined as $L_t^* = \{\boldsymbol{w} \in \mathbb{R}^t \mid \boldsymbol{w} \cdot \boldsymbol{v} \in \mathbb{Z} \text{ for all } \boldsymbol{v} \in L_t\}$. The *dual* of a given basis $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ is the set of vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_t$ in $\mathbb{R}^t$ such that $\boldsymbol{v}_i \cdot \boldsymbol{w}_j = \delta_{ij}$, where $\delta_{ij} = 1$ if $i = j$, and $\delta_{ij} = 0$ otherwise. The vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_t$ form a basis of the dual lattice. These $\boldsymbol{w}_j$'s are the columns of the matrix $\boldsymbol{V}^{-1}$, the inverse of the matrix $\boldsymbol{V}$. We define $\boldsymbol{W} = (\boldsymbol{V}^{-1})^{\mathsf{t}}$, so that the *rows* of $\boldsymbol{W}$ form the dual basis. To *dualize* a lattice means interchanging $\boldsymbol{V}$ for $\boldsymbol{W}$, i.e., when switching between the primal and dual. The density η_t^* of L_t^* is the inverse of the density of L_t; that is, $1/\det(L_t^*) = \det(L_t) = \det(\boldsymbol{V})$ and $\eta_t^* = 1/\eta_t$.

The dual lattice is a key tool to study the uniformity of a lattice point set for the following reason. For any vector $\boldsymbol{w} \in L_t^*$ and any integer z, the set $\{\boldsymbol{v} \in \mathbb{R}^s : \boldsymbol{w} \cdot \boldsymbol{v} = z\}$ is a hyperplane orthogonal to $\boldsymbol{w}$. When z goes through all the integers in $\mathbb{Z}$, these sets form a family of equidistant parallel hyperplanes that cover all points of L_t, because $\boldsymbol{v} \cdot \boldsymbol{w} \in \mathbb{Z}^t$ for all $\boldsymbol{v} \in L_t$. If $\boldsymbol{w}$ is an integer multiple of another dual lattice vector, this remains true, although the family will contain more hyperplanes than necessary to contain all the points. The Euclidean distance between successive hyperplanes in this family is the distance between the origin and the hyperplane with $z = 1$, i.e., the length of the shortest nonzero vector from the origin to that hyperplane. This length turns out to be $1/\|\boldsymbol{w}\|_2$, the inverse of the length of the dual vector $\boldsymbol{w}$ [25, 39]. Therefore, if we compute the Euclidean length of the shortest nonzero vector in L_t^* and take the inverse of this length, we get the distance between successive hyperplanes for the family for which this distance is the largest.

Likewise, the length of the shortest nonzero vector in L_t^* with the L^1 norm, minus 1, is equal to the minimal number of parallel hyperplanes required to cover $L_t \cap (0,1)^t$, the set of lattice points that lie in the open unit hypercube [67, 13]. If we insist on covering also the point $\mathbf{0}$, i.e., $L_t \cap [0,1)^t$, then we need one extra hyperplane when all coordinates of $\boldsymbol{w}$ have the same sign [25, Exercise 3.3.4-16].

Because of these properties, computing shortest nonzero vectors in dual lattices with the L^2 or L^1 norms is an important task. We want these shortest vectors to be as long as possible, given the constraints we have on the lattice density.

3.2 Rescaling the primal and dual to integral lattices

In Lattice Tester, it is assumed that all basis vector coordinates are integers, so they can be represented exactly on a computer. In case the basis vectors are rational, it is always possible to find positive integers m_1 and m_2 such that $\Lambda_t = m_1 L_t \subseteq \mathbb{Z}^t$ and $\Lambda_t^* = m_2 L_t^* \subseteq \mathbb{Z}^t$, then we

can work with the rescaled lattice Λ_t and its *m-dual* rescaled lattice Λ_t^*, where $m = m_1 m_2$. That is, we rescale either $\boldsymbol{V}$ or $\boldsymbol{W}$ or both by the integer factors m_1 and m_2. The densities of the rescaled lattice Λ_t and of its m-dual Λ_t^* are then related by $m^t/\det(\Lambda_t^*) = \det(\Lambda_t) = m_1^t \det(L_t)$. For all the applications that we are using this software for, L_t^* is already an integral lattice, and only L_t is rescaled by m. This rescaling divides the density by m^t in t dimensions. We will denote by Λ_I the projection of Λ_t over a subset I of coordinates, and by Λ_I^* its m-dual.

In the software and in much of the remainder of this document, we assume that the appropriate rescaling has already been done and we work directly with the rescaled lattice Λ_t, and its m-dual $\Lambda_t^* = L_t^*$, which are both $\mathbb{Z}$-lattices over $\mathbb{Z}^t$. Having integral lattices permits us to always represent vector coordinates exactly on the computer. From now on, we will use $\boldsymbol{V}$ with rows $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ to represent a basis of Λ_t (and not of L_t as in the previous section) and $\boldsymbol{W}$ with rows $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_t$ for a basis of its m-dual lattice Λ_t^*. We say that $\boldsymbol{V}$ and $\boldsymbol{W}$ are *m-dual* to each other.

Since $\boldsymbol{V} \cdot \boldsymbol{W}^{\mathsf{t}} = m\boldsymbol{I}$ and all the entries of $\boldsymbol{V}$ and $\boldsymbol{W}$ are integers, each vector $m\boldsymbol{e}_i$ (m times the ith *unit vector* $\boldsymbol{e}_i$) can be expressed as an integer linear combination of the basis vectors in $\boldsymbol{V}$ (equivalently, L_t contains all integer vectors) and also as an integer linear combination of the basis vectors in $\boldsymbol{W}$. Therefore, each vector $m\boldsymbol{e}_i$ must belong to both the primal and m-dual lattices, for $i = 1, \ldots, t$. This means that any linear combination of lattice vectors modulo m gives another lattice vector, and similarly for the m-dual. Therefore, all operations on vectors to construct or reduce a basis (see the following sections) can be performed modulo m, as long as the mod m operation does not exclude a vector $m\boldsymbol{e}_i$ from the lattice. To avoid such exclusion, we can always assume when building a basis that the vectors $m\boldsymbol{e}_i$ belong implicitly to the set of generating vectors. The fact that all vectors $m\boldsymbol{e}_i$ belong to the lattice also implies that any basis for the projection L_I over a subset I of coordinates must contain $|I|$ independent vectors, so it must be a square matrix, and similarly for its m-dual. This property is convenient when studying the uniformity of projections.

For a $\mathbb{Z}$-lattice with basis $\boldsymbol{V}$, the m-dual basis $\boldsymbol{W}$ can be computed for *any* integer $m \geq 1$, but the entries of $\boldsymbol{W}$ are not necessarily all integers for all m. This holds only if the lattice L_t generated by $\boldsymbol{V}/m$ contains $\mathbb{Z}^t$, or equivalently if all vectors $m\boldsymbol{e}_i$ are integer linear combinations of the rows of $\boldsymbol{V}$. In the applications targeted by this software, this condition is satisfied. The software assumes that the condition holds and takes advantage of this. We will work sometimes only with $\boldsymbol{V}$, sometimes only with $\boldsymbol{W}$, and sometimes with both.

As a special case, we have a *rescaled integration lattice of rank 1* [12, 50, 81] if the rows of $\boldsymbol{V}$ have the form $\boldsymbol{v}_1 = \boldsymbol{a} = (a_1, a_2, \ldots, a_t)$ for some integer vector $\boldsymbol{a}$ whose first coordinate a_1 divides m, and $\boldsymbol{v}_2 = m\boldsymbol{e}_2, \ldots, \boldsymbol{v}_t = m\boldsymbol{e}_t$ complete the basis, where m is a positive integer. For such a lattice, it is easy to enumerate the lattice points that lie in the rescaled unit hypercube $[0, m)^t$ and then map them to their unscaled versions in $[0, 1)^t$: the rescaled points are simply $i\boldsymbol{a} \bmod m$ for $i = 0, 1, \ldots, m-1$. They are distinct if and only if at least one coordinate of $\boldsymbol{a}$ is relatively prime to m. The projection Λ_I of Λ_t over a subset I of coordinates has m distinct points for all nonempty subsets I if and only if all coordinates of $\boldsymbol{a}$ are relatively prime to m. The latter condition is usually imposed in applications.

When the lattice L_t corresponds to vectors of t successive outputs from a linear *multiple recursive generator (MRG)* of order k with modulus m [30, 32, 46] and $t \geq k$, one has $\Lambda_t = mL_t$, $\det(L_t) = m^{-k}$, $\det(\Lambda_t) = m^{t-k}$ (which gets very large when t gets large), and $\det(L_t^*) = \det(\Lambda_t^*) = m^k$. When L_t comes from a rank-1 lattice rule with m points, we have this with $k = 1$. Note that $\det(\Lambda_t)$ grows exponentially with t and may not be easily representable on the computer for very large m and t. In the software, we avoid computing it (or its multiplicative inverse, the density) directly for this reason. We compute and use its logarithm instead.

For the projection I of an MRG-based lattice over the first s coordinates with $s < k$, we have $\det(L_I) = m^{-s}$, $\det(\Lambda_I) = 1$, and $\det(L_I^*) = \det(\Lambda_I^*) = m^s$. For a projection I over $s < k$ non-successive outputs, we may have $\det(L_I) > m^{-s}$ and $\det(L_I^*) = \det(\Lambda_I^*) < m^s$. We have the strict inequality when points are projected over each other, so the density of L_I becomes smaller than m^s. There is an example of this in the LatMRG guide [40]. [1]

3.3 Lengths of shortest nonzero vectors in a lattice and its projections

For the applications targeted by this software, the lattices and their projections are defined by first constructing the rescaled primal lattice Λ_t, sometimes taking projections Λ_I or adding coordinates for this primal lattice, and then computing the m-dual of the lattice or of the projection. If $\boldsymbol{w}$ is in the m-dual L_t^* and we add one coordinate to the lattice to get Λ_{t+1}, then it suffices to add a zero coordinate to $\boldsymbol{w}$ and this new vector will be in the m-dual L_{t+1}^*. This means that when we add a new coordinate, the length of the shortest nonzero m-dual vector cannot increase, it can only decrease or remain the same. By the same argument, the length of a shortest nonzero vector in the m-dual of a projection Λ_I of Λ_t cannot be smaller than the one in L_t^*. This property does not apply to the primal lattice. Things are not symmetric because we are not making direct projections of the m-dual. In the primal lattice, the shortest vector lengths can only increase when we add coordinates, because if $\boldsymbol{v}$ is a shortest nonzero vector in Λ_t, then the projection of $\boldsymbol{v}$ on the coordinates in I belongs to Λ_I and cannot be longer than $\boldsymbol{v}$. So the shortest vector length cannot increase when we remove coordinates and cannot decrease when we add coordinates.

What typically happens is that when t increases, at some point the length of the shortest nonzero vector in Λ_t becomes m and each $m\boldsymbol{e}_i$ is a shortest nonzero vector. In particular, suppose that $\Lambda_t \cap [0, m)^t$ contains m points and that its projection on each single coordinate is $\mathbb{Z}_m = \{0, \ldots, m-1\}$. This implies that none of the $m-1$ nonzero points in $\Lambda_t \cap [0, m)^t$ has a zero coordinate. In that case, a shortest nonzero vector can be at a corner of the hypercube $[0, m]^t$, or otherwise all of its coordinates must be nonzero. But in the latter case, its Euclidean length must be at least $\sqrt{t}$, and therefore $m\boldsymbol{e}_i$ is necessarily a shortest nonzero vector whenever $t \geq m^2$. Often, this occurs already for a much smaller dimension t_0, and then for all $t \geq t_0$.

[1] From Pierre: Give specific example number when that guide is ready.

As a simple worst-case example, suppose a basis of L_t is $(1/m, \ldots, 1/m)$ and the unit vectors $\boldsymbol{v}_i = \boldsymbol{e}_i$ for $i \geq 2$, so a basis for Λ_t is $\boldsymbol{v}_1 = (1, \ldots, 1)$ and $\boldsymbol{v}_i = m\boldsymbol{e}_i$ for $i \geq 2$. Then for the L^2 norm, $\boldsymbol{v}_1$ is a shortest vector of length $\sqrt{t}$ for $t \leq m^2$ and any $\boldsymbol{e}_i$ is a shortest vector of length m for $t \geq m^2$. For the L^1 norm $\boldsymbol{v}_1$ is a shortest vector of length t for $t \leq m$. Note that all the points of $\Lambda_t \cap [0, m)^t$ are on a single line in this case.

As another example, if $m = 5$, $\boldsymbol{v}_1 = (1, 2, 3, 4, \ldots)$, and $\boldsymbol{v}_i = m\boldsymbol{e}_i$ for $i \geq 2$, then the only lattice points that belong to $(0, 1)^t$ are $j\boldsymbol{v}_1 \bmod m$ for $j = 1, \ldots, m-1$. For $t = 4$, these points are $(1, 2, 3, 4)$, $(2, 4, 1, 3)$, $(3, 1, 4, 2)$, $(4, 3, 2, 1)$. They all have an L^2 norm of $\sqrt{30} > m$, so the shortest vectors in 4 dimensions or more are the vectors $m\boldsymbol{e}_i$, of length m, regardless of the other coordinates of $\boldsymbol{v}_1$. For the L^1 norm, this occurs already in 3 dimensions.

Larger examples are examined in Section 12.6.

4 Problems of Interest with Integral Lattices

In this document (and software), we are particularly interested in the following problems for integral lattices over $\mathbb{Z}$.

A. **Lattice Basis Construction**. Given a set of vectors (not necessarily independent) with integer coordinates, find a basis for the lattice generated by these vectors. Note that our representation of a lattice on the computer is by a lattice basis, so the first step is always to construct such a basis.

B. **Find m-Dual Lattice Basis**. Given a lattice basis, compute the corresponding m-dual basis.

C. **Lattice Basis Reduction (LBR).** Given a lattice basis, find another basis whose vectors are nearly orthogonal or as short as possible in some sense. There are many variants and definitions of this.

D. **Shortest Vector Problem (SVP).** Find a shortest nonzero vector in the lattice, with a proof that it is shortest.

E. **Approximate Shortest Vector Problem (ASVP).** Find a short vector that is not much larger than a shortest nonzero vector in the lattice.

The next sections explain how we can solve each of these problems and how it is implemented in Lattice Tester. Problems A, B, and E are relatively easy, whereas D is much harder. For C, there are many ways of defining the concept of reduced basis and the difficulty depends on this definition. For example, LLL-reduction (a weak form of basis reduction) takes only polynomial time, whereas Minkowski and Korkine-Zolotarev reduction take exponential time in the dimension, with the currently available algorithms [5, 70]. The best known algorithms for Problem D take exponential time as a function of the dimension, but

in practice we can solve reasonably large instances, particularly in situations where a very short vector can be identified very quickly.

In Problems B to D, vector lengths are usually measured with the Euclidean (L^2) norm and this is what we assume in the rest of this manual, except when indicated otherwise. Sometimes we are interested in other norms. In particular, the software handles the L^1 norm for Problems D and E.

5 Building a Basis from a Set of Generating Vectors

5.1 Context and motivation

In many cases, a lattice basis and its m-dual are obtained directly from the problem specification, so there is no Problem A to solve. This occurs for example for the lattice that corresponds to successive output values produced by a multiple recursive generator [25, 30, 47]. But in some situations, we need to extract a basis from a set of possibly dependent generating vectors.

Given a finite set of (nonzero) integer vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s \in \mathbb{Z}^t$ that are not necessarily independent, we want to construct a basis for the lattice generated by these vectors together with the vectors $m\boldsymbol{e}_i$ in t dimensions. We may also want to obtain the corresponding m-dual basis. Here, t is the dimension of the vectors, s can be smaller, equal, or larger than t. If some of these vectors are zero, we can simply remove them and work with those that remain. In our setting, we know a priori that the vectors $m\boldsymbol{e}_i$ for $i = 1, \ldots, t$ must belong to the lattice, so we always add them to the set of generating vectors, often implicitly. This implies that the generated lattice must have t dimensions.

One important situation in which this basis construction problem occurs is when we need to construct a basis for the projected rescaled lattice Λ_I over the subset of coordinates $I \subset \{1, \ldots, t\}$, given a basis for Λ_t. By projecting the vectors of Λ_t over the s coordinates in I, with $s < t$, we obtain a set of dependent s-dimensional vectors that, together with the vectors $m\boldsymbol{e}_i$, generate Λ_I. We typically want to use this set to build a basis for Λ_I and perhaps the corresponding m-dual basis. Both of these bases will be $s \times s$ invertible matrices. Another common situation is when we already have a basis for Λ_I, and we want to add one coordinate to the set I, or replace one coordinate of I by a different one, and we want to recover a basis for the modified Λ_I and perhaps its m-dual as well.

Given a set $\mathcal{S}$ of generating vectors for a lattice, if a new set $\mathcal{S}'$ is obtained from $\mathcal{S}$ by applying any of the following operations, then $\mathcal{S}$ and $\mathcal{S}'$ generate the same lattice. The operations allowed are: (1) change the sign of a vector (multiply it by -1); (2) add (or subtract) an integer multiple of one vector to another one; (3) remove a vector if it is zero. Multiplying a vector by a constant other than ± 1 is not allowed. In case $\mathcal{S}$ already corresponds to the rows of a basis matrix $\boldsymbol{V}$, one can permute the rows of $\boldsymbol{V}$, multiply a row by -1, or add an integer multiple of one row to another one. Each of these operations

corresponds to multiplying $\boldsymbol{V}$ by a unimodular matrix, so it only transforms $\boldsymbol{V}$ into another basis of the same lattice.

5.2 The gcd triangular construction method

We say that we have an upper-triangular basis of Λ_t when the matrix $\boldsymbol{V}$ is upper triangular, and a lower-triangular basis when it is lower triangular. In both cases, the determinant of the lattice is the product of the diagonal elements: $\det(\Lambda_t) = v_{1,1} \cdots v_{t,t}$. One advantage of having a triangular basis is that it is very easy to compute its m-dual; see Section 6. We now describe algorithms to compute an upper or lower triangular basis for the lattice generated by $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s$ together with the vectors $m\boldsymbol{e}_1, \ldots, m\boldsymbol{e}_t$ which are added implicitly to form a set of generating vectors. At the end, the vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$ will form a triangular basis whose diagonal elements are all positive.

Our first algorithm will return an *upper-triangular basis*. We have s vectors in t dimensions. For the simplest case where the s vectors are in one dimension ($t = 1$), these vectors are just s integers denoted $v_1, \ldots, v_s$. We first replace each v_j by $v_j \bmod m$, so we now have $0 \le v_j < m$ for all j. Then we compute $c = \gcd(v_1, \ldots, v_s, m) > 0$, the greatest (positive) common divisor of these s integers and m. When doing this, we skip the v_j's that are zero. This c belongs to the lattice and every lattice point must be an integer multiple of c, so c is a lattice basis. This gcd can be computed via Euclid's algorithm as follows: start with $c = m$ and for $j = 1, \ldots, s$, if $v_j \neq 0$, compute $c := \gcd(c, v_j)$. If all v_j's are 0, we get $c = m$.

For $t > 1$, we compute an upper-triangular basis as follows. Let $\boldsymbol{v}_i = (v_{i,1}, \ldots, v_{i,t})$ for $i = 1, \ldots, s$. We first replace each $v_{i,j}$ by $v_{i,j} \bmod m$, to obtain a set of s generating vectors with $0 \le v_{i,j} < m$ for all (i, j), together with the vectors $m\boldsymbol{e}_i$. Let $c_1 = \gcd(v_{1,1}, \ldots, v_{s,1}, m)$, assuming here that the zero values are skipped. If $c_1 = m$, which means that $v_{i,1} = 0$ for all i, we take $\boldsymbol{x}_1 = m\boldsymbol{e}_1$ as our first basis vector. Otherwise, we exploit the fact that c_1 can be written as $c_1 = a_{0,1}m + a_{1,1}v_{1,1} + \cdots + a_{s,1}v_{s,1}$ for some integers $a_{i,1} \in \mathbb{Z}$, $-m < a_{i,1} < m$, that can be computed while doing Euclid's algorithm. Then the lattice vector $\boldsymbol{x}_1 = a_{1,1}\boldsymbol{v}_1 + \cdots + a_{s,1}\boldsymbol{v}_s \bmod m$ has $c_1 > 0$ as its first coordinate, and this is the first vector of our triangular basis. Since this c_1 divides each nonzero $v_{i,1}$, we can put $\boldsymbol{v}_i = \boldsymbol{v}_i - (v_{i,1}/c_1)\boldsymbol{x}_1 \bmod m$ for $i = 1, \ldots, s$, so all these generating vectors now have 0 as their first coordinate, and their other coordinates are in $\mathbb{Z}_m = \{0, \ldots, m-1\}$. The new set $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s, \boldsymbol{x}_1, m\boldsymbol{e}_2, \ldots, m\boldsymbol{e}_t$ generates the same lattice as before. Note that $m\boldsymbol{e}_1$ is no longer needed, since it can be obtained as an integer linear combination of the other vectors. Here we compute the vector $\boldsymbol{x}_1$ only after computing c_1, and we modify the vectors $\boldsymbol{v}_i$ only after computing $\boldsymbol{x}_1$. This is more efficient than updating $\boldsymbol{x}_1$ while computing the gcd, especially when t is large, because it could save a few vector operations if some coefficients $a_{i,1}$ are 0. On the other hand, the cost of updating the $\boldsymbol{v}_i$'s is the same both ways.

Then we repeat the same process with the new vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s$, but using their second column. If $v_{i,2} = 0$ for $i = 1, \ldots, s$, we put $\boldsymbol{x}_2 = m\boldsymbol{e}_2$. Otherwise, we compute $c_2 = \gcd(v_{1,2}, \ldots, v_{s,2}, m) = a_{1,2}v_{1,2} + \cdots + a_{s,2}v_{s,2} \bmod m$, we put $\boldsymbol{x}_2 = a_{1,2}\boldsymbol{v}_1 + \cdots + a_{s,2}\boldsymbol{v}_s \bmod m$

as our second basis vector, and we use $\boldsymbol{x}_2$ to zeroes the nonzero second coordinates of the $\boldsymbol{v}_i$'s via $\boldsymbol{v}_i = \boldsymbol{v}_i - (v_{i,2}/c_2)\boldsymbol{x}_2 \bmod m$ for all i.

Then we do the same for the third column, and so on, until we have $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$. At that point, all the $\boldsymbol{v}_i$ will be zero and we will no longer need any of the $m\boldsymbol{e}_i$. All these operations preserve the property that after each step j, the vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s$ together with $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_j$ that have been computed so far and $m\boldsymbol{e}_{j+1}, \ldots, m\boldsymbol{e}_t$, are a set of generating vectors for our lattice. At the end, the $\boldsymbol{v}_i$ are all zero, we no longer need the $m\boldsymbol{e}_i$'s, and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$ form an upper-triangular invertible matrix which gives an upper triangular lattice basis.

Observe that after we put all the elements $v_{i,j}$ of column j to 0 by doing $\boldsymbol{v}_i = \boldsymbol{v}_i - (v_{i,j}/c_j)\boldsymbol{x}_j \bmod m$ for all i, none of these elements will be used again in any way. Therefore we can save computations by performing this subtraction only for the columns $j+1$ to t, leaving "garbage" in the previous columns. That is, we just do $v_{i,k} = v_{i,k} - (v_{i,j}/c_j)x_{j,k} \bmod m$ for $k = j+1, \ldots, t$ and $i = 1, \ldots, s$.

To compute the gcd and the coefficients $a_{i,j}$, for the first column we proceed incrementally by first putting $g_0 = m$, and then computing $g_i = \gcd(g_{i-1}, v_{i,1})$ if $v_{i,1} \neq 0$ and $g_i = g_{i-1}$ otherwise, for $i = 1, \ldots, s$. At the end we have $c_1 = g_s$. Each time we compute $\gcd(g_{i-1}, v_{i,1})$, Euclid's algorithm also returns two integers (a'_{i-1}, a_2) such that $g_i = a'_{i-1}g_{i-1} + a_i v_{i,1}$, for $i = 1, \ldots, s$. With easy calculations, this gives us the integers $a_{0,1}, a_{1,1}, \ldots, a_{s,1}$ such that $c_1 = g_s = a_{1,1}v_{1,1} + \cdots + a_{s,1}v_{s,1}$. We use the same process for all the other columns.

In what we have just described, the coordinates of the vectors $\boldsymbol{x}_i$ that are above the diagonal are reduced modulo m in a way that they are never negative. To make the basis vectors shorter, it is also possible to reduce these coordinates so they lie in $\{-m/2+1, \ldots, m/2\}$ instead of $\{0, \ldots, m-1\}$. We call this *modulo m reduction toward zero* and we denote it by "$\bmod_0$". Starting with shorter basis vectors is generally better when we apply the BB algorithm of section 8.5 and also when reducing a basis with LLL. For this reason, we use this reduction toward zero in our implementation.

The complete procedure is stated in Algorithm 1.

To compute a *lower-triangular basis*, we proceed in the same way but starting from the last column and lower right corner instead of the first column and upper left corner. We first replace each $\boldsymbol{v}_i$ by $\boldsymbol{v}_i \bmod m$. Then we compute the gcd c_t between the last coordinates that are nonzero, and m; that is, $c_t = \gcd(v_{1,t}, \ldots, v_{s,t}, m) = a_{0,t}m + a_{1,t}v_{1,t} + \cdots + a_{s,t}v_{s,t}$. If $c_t = m$, we take $\boldsymbol{x}_t = m\boldsymbol{e}_t$ as our last basis vector, otherwise we take $\boldsymbol{x}_t = a_{1,t}\boldsymbol{v}_1 + \cdots + a_{s,t}\boldsymbol{v}_s \bmod_0 m$. This last vector has $c_t > 0$ as its last coordinate. Since c_t divides each nonzero $v_{i,t}$, we can put $\boldsymbol{v}_i = \boldsymbol{v}_i - (v_{i,t}/c_t)\boldsymbol{x}_t \bmod m$ for $i = 1, \ldots, s$, so all our previous generating vectors now have zero as their last coordinate. The new set $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s, m\boldsymbol{e}_1, \ldots, m\boldsymbol{e}_{t-1}, \boldsymbol{x}_t$ generates the same lattice as before.

Then, we repeat the same process with the new vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s$, but using their next-to-last column. We compute $c_{t-1} = \gcd(v_{1,t-1}, \ldots, v_{s,t-1}, m) = a_{0,t-1}m + a_{1,t-1}v_{1,t-1} + \cdots + a_{s,t-1}v_{s,t-1} \bmod m$. If $c_{t-1} = m$, we take $\boldsymbol{x}_{t-1} = m\boldsymbol{e}_{t-1}$, otherwise we put $\boldsymbol{x}_{t-1} = a_{1,t-1}\boldsymbol{v}_1 + \cdots + a_{s,t-1}\boldsymbol{v}_s \bmod_0 m$. We use $\boldsymbol{x}_{t-1}$ to zeroes all nonzero next-to-last coordinates of the $\boldsymbol{v}_i$'s via $\boldsymbol{v}_i = \boldsymbol{v}_i - (v_{i,t-1}/c_{t-1})\boldsymbol{x}_{t-1} \bmod m$ for all i. Then we do the same for column

Algorithm 1: Computing an upper-triangular basis from generating vectors

UT$(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s)$:
// $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s, m\boldsymbol{e}_1, \ldots, m\boldsymbol{e}_t$ are assumed to be generating vectors
for $i = 1, \ldots, s$ **do**
  $\boldsymbol{v}_i \leftarrow \boldsymbol{v}_i \bmod m$;
for $j = 1, \ldots, t$ **do**
  $c_j \leftarrow \gcd(v_{1,j}, \ldots, v_{s,j}, m) = a_{0,j}m + a_{1,j}v_{1,j} + \cdots + a_{s,j}v_{s,j}$;
  if $c_j = m$ **then**
    $\boldsymbol{x}_j = m\boldsymbol{e}_j$
  else
    $\boldsymbol{x}_j = a_{1,j}\boldsymbol{v}_1 + \cdots + a_{s,j}\boldsymbol{v}_s \bmod_0 m$;
    for $i = 1, \ldots, s$ **do**
      for $k = j+1, \ldots, t$ **do**
        $v_{i,k} = v_{i,k} - (v_{i,j}/c_j)x_{j,k} \bmod m$.

$t-2$, and so on, until we have $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$. All these operations preserve the property that the vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s$ together with $m\boldsymbol{e}_1, \ldots, m\boldsymbol{e}_{j-1}, \boldsymbol{x}_j, \ldots, \boldsymbol{x}_t$ are always a set of generating vectors for our lattice. At the end, the $\boldsymbol{v}_i$ are all zero and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$ form a lower-triangular invertible matrix which gives an lower triangular lattice basis.

Again, when doing $\boldsymbol{v}_i = \boldsymbol{v}_i - (v_{i,j}/c_j)\boldsymbol{x}_j \bmod m$ for all i, to save time we just do $v_{i,k} = v_{i,k} - (v_{i,j}/c_j)x_{j,k} \bmod m$ for $k = 1, \ldots, j-1$ and $i = 1, \ldots, s$, because the other elements $v_{i,k}$ will never be used again.

This procedure is stated in Algorithm 2.

Algorithm 2: Computing a lower-triangular basis from generating vectors

LT$(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s)$:
// $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s, m\boldsymbol{e}_1, \ldots, m\boldsymbol{e}_t$ are assumed to be generating vectors
for $i = 1, \ldots, s$ **do**
  $\boldsymbol{v}_i \leftarrow \boldsymbol{v}_i \bmod m$;
for $j = t, \ldots, 1$ **do**
  $c_j \leftarrow \gcd(v_{1,j}, \ldots, v_{s,j}, m) = a_{0,j}m + a_{1,j}v_{1,j} + \cdots + a_{s,j}v_{s,j}$;
  if $c_j = m$ **then**
    $\boldsymbol{x}_j = m\boldsymbol{e}_j$
  else
    $\boldsymbol{x}_j = a_{1,j}\boldsymbol{v}_1 + \cdots + a_{s,j}\boldsymbol{v}_s \bmod_0 m$;
    for $i = 1, \ldots, s$ **do**
      for $k = 1, \ldots, j-1$ **do**
        $v_{i,k} = v_{i,k} - (v_{i,j}/c_j)x_{j,k} \bmod m$.

The two algorithms just described for the upper-triangular and lower-triangular constructions are implemented in the `BasisConstruction.h` file.

A similar algorithm to compute an upper-triangular basis is briefly described in Section 7 of [10] and in Section 3 of [47], and was implemented in the Modula-2 version of LatMRG. One difference is that instead of computing each gcd's c_i before modifying any vector, the old algorithm modifies the vectors along the way while computing the gcd. The code was also written in a different way. The old method is also offered in `BasisConstruction.h`, for testing and comparisons. Our newer versions are much faster, as we will see in Section 12.5.

5.3 The LLL construction method

NTL uses a different approach to build a basis with a modified LLL procedure that identifies and eliminates along the way the vectors that can be expressed as integer linear combinations of the other ones. Instead of constructing a triangular basis, it tries to reduce the length of each vector as much as possible by subtracting a multiple of another vector (a pairwise reduction) having a smaller index, and exchanging vectors whenever a vector becomes significantly shorter than its predecessor, as in the LLL reduction. The algorithm is stated in [5, Page 109]. See also [22, 72]. It provides a basis that in general is *not* triangular, but is comprised of shorter vectors than the procedures of Section 5.2. This approach is implemented in the LLL functions of NTL, which are accessible from our `ReducerStatic.h` file. Instead of giving a basis as input to these functions, one can just give a set of generating vectors.

Speed comparisons between LLL and triangular basis constructions are reported in Sections 12.5 and 12.4. In general, if the end goal is to obtain an *LLL-reduced primal basis*, it is slightly more effective to use the LLL construction directly rather than construct a triangular basis and apply LLL afterwards. But if the end goal is to obtain an *LLL-reduced m-dual basis*, it is more effective to construct a triangular basis, then compute its (triangular) m-dual, and apply LLL to it. The reason is that computing the m-dual is significantly faster in the triangular case (see Section 6). Also, doing the LLL construction or reduction before applying the gcd triangular method does not improve the speed of the latter.

5.4 Direct construction for a rank-1 integration lattice

For a rank-1 integration lattice with generating vector $\boldsymbol{a} = (1, a_2, \ldots, a_t)$ in t dimensions (we always take $a_1 = 1$ for simplicity), a basis for $\Lambda_t = mL_t$ is formed by the rows $\boldsymbol{v}_i$ of the matrix

$$\boldsymbol{V} = \begin{pmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \vdots \\ \boldsymbol{v}_t \end{pmatrix} = \begin{pmatrix} 1 & a_2 & \ldots & a_t \\ 0 & m & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & \ldots & m & 0 \\ 0 & \ldots & 0 & m \end{pmatrix} \tag{2}$$

whose m-dual is

$$\boldsymbol{W} = \begin{pmatrix} \boldsymbol{w}_1 \\ \boldsymbol{w}_2 \\ \vdots \\ \boldsymbol{w}_t \end{pmatrix} = \begin{pmatrix} m & 0 & \dots & 0 & 0 \\ -a_2 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -a_{t-1} & 0 & \dots & 1 & 0 \\ -a_t & 0 & \dots & 0 & 1 \end{pmatrix}. \tag{3}$$

For the case of a Korobov lattice, we have $a_j = a^{j-1} \bmod m$ for all j, for some integer $a > 0$. This also applies to an LCG with modulus m and multiplier a, whose successive states follow the recurrence $x_n = ax_{n-1} \bmod m$, so any vector of t successive states must be an integer multiple of the vector $(1, a, a^2, \dots, a^{t-1})$ modulo m.

Suppose we have constructed a basis in $t = s - 1$ dimensions, we reduced it in some way (e.g., by LLL or otherwise), and we now want to increase its dimension to $t = s$. We can of course just naively reconstruct the basis and its m-dual in s dimensions directly as above. But if s is large and the basis in $s - 1$ dimensions was already reduced to a basis with short vectors, we may not want to lose this previous reduction work and prefer to expand the current basis to account for the new coordinate. This can be done as follows.

We need to add one row and one column to the current basis matrices $\boldsymbol{V}$ and $\boldsymbol{W}$ (which differ from those written above). Let $v_{i,j}$ and $w_{i,j}$ denote the current elements of these matrices. The new row of $\boldsymbol{V}$ can simply be $m\boldsymbol{e}_s = (0, \dots, 0, m)$. For the new column, notice that $\boldsymbol{v}_i$ in the new basis must be equal modulo m to $v_{i,1}$ times the vector $(1, a_2, \dots, a_s)$, because it must be a linear combination modulo m of the initial basis vectors. Therefore, we must have $v_{i,s} = a_s v_{i,1} \bmod m$ for all i. This gives the new basis.

In case we have reduced versions of both the primal and m-dual bases in $s-1$ dimensions and we want to extend both of them to s dimensions, it suffices to add a zero coordinate to each vector of the current m-dual basis, then add the vector

$$\boldsymbol{w}_s = \boldsymbol{e}_s - \frac{1}{m} \sum_{i=1}^{s-1} v_{i,s} \boldsymbol{w}_i,$$

where the $\boldsymbol{w}_i$ are the previous m-dual basis vectors. One can verify that we indeed have $\boldsymbol{V}\boldsymbol{W}^{\mathrm{t}} = m\boldsymbol{I}$ with the new matrices. With this approach, one never has to compute the m-dual basis by inverting the primal one, but the m-dual basis must be maintained whenever we perform reduction operations on the primal, and vice-versa. The LLL implementation from NTL does not do that.

The previous construction requires that we update either only the primal, or both the primal and m-dual bases together. In case we want to maintain only the m-dual and not the primal, a simple alternative is to add a zero coordinate to each vector of the current basis, then we add the vector

$$\boldsymbol{w}_s = \boldsymbol{e}_s - a_s \boldsymbol{e}_1 = (-a_s, 0, \dots, 0, 1).$$

This provides a basis for the m-dual lattice and this construction is certainly faster than the previous one. In both cases, the first $s-1$ vectors of the m-dual basis are already reduced and

we only need to update the reduction to account for the new vector. The difference between the two choices of $\boldsymbol{w}_s$ is only a sum of vectors that were already in the $(s-1)$-dimensional lattices, so they just give two different bases for the same lattice.

Suppose now that we are given a subset $I \subset \{1, \ldots, t\}$ and we want to construct a basis for Λ_I or Λ_I^*. When we analyze the lattice structure of recurrence-based RNGs, it suffices to consider only subsets I that contain the first coordinate, i.e., $1 \in I$, because all other projections are identical to such projections, due to the projection-stationary property (see Section 10). In this case where $1 \in I$, we can just select in $\boldsymbol{V}$ the columns are rows whose indices are in I, and this forms an upper-triangular basis of Λ_I, because when we remove the columns that are not in I, the rows whose numbers are not in I become zero. It also suffices to select in $\boldsymbol{W}$ the rows and columns whose indices are in I and they form a lower-triangular basis of the m-dual lattice Λ_I^*. If $s = |I|$, then $\det(\Lambda_I) = m^{s-1}$ and $\det(\Lambda_I^*) = m$. Thus, triangular bases for these projections and their m-duals can be found directly, without any computation.

When the first coordinate does not belong to I, on the other hand, after selecting the columns of $\boldsymbol{V}$ whose numbers are in I, we obtain $s = |I|$ columns and $s+1$ nonzero rows, which we have to reduce to obtain a basis. If we only want a basis for the primal lattice, we can use LLL for that. If we want a basis for the m-dual lattice, then it is usually better to first compute an upper-triangular basis for the primal, and then compute the m-dual of that basis. Some functions in `BasisConstruction.h` do that.

If $I = \{i_1, \ldots, i_s\}$, the $s+1$ nonzero rows are the rows of the matrix

$$\begin{pmatrix} a_{i_1} & \ldots & a_{i_s} \\ m & \ldots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ldots & m \end{pmatrix}.$$

If we apply the lower-triangular gcd method of Section 5.2 to this matrix, the first step will be to compute $c_1 = \gcd(a_{i_1}, m) = b_1 a_{i_1} + b_2 m$ by Euclid's algorithm, and put $\boldsymbol{x}_1 = b_1 \boldsymbol{v}_1 + b_2 \boldsymbol{v}_2 \bmod m$, where $\boldsymbol{v}_1 = (a_{i_1}, \ldots, a_{i_s})$ and $\boldsymbol{v}_2 = m\boldsymbol{e}_1$ are the first two rows of the above matrix. The vector $\boldsymbol{x}_1 = (x_1, \ldots, x_s)$ has $x_1 = c_1$ as its first coordinate, and its other coordinates are $x_j = b_1 a_{i_j} \bmod m$ for $j = 2, \ldots, s$. Then we replace $\boldsymbol{v}_2$ by $\boldsymbol{v}_2 - (m/c_1)\boldsymbol{x}_1 \bmod m$, so its first coordinate becomes 0.

Recall that c_1 must be a divisor of m. In particular, we always have $c_1 = 1$ if m is prime, or if m is a power of 2 and a_1 is odd. For all good LCG or lattice rule constructions, we should have $c_1 = 1$. If $c_1 = 1$, then the new $\boldsymbol{v}_2$ is zero and an m-dual basis can be computed directly, exactly as in (3), with x_j in place of a_j. For the lattices coming from RNGs and QMC, we should always have $c_1 = 1$. If $c_1 > 1$, then the first coordinate of any lattice vectors must be a multiple of c_1 so the first coordinate can only take m/c_1 distinct values. This means that the projection over that first coordinate is not regular, it loses points. The next example illustrates this case.

Example 1. Let $m = 8$, $t = 3$, and $\boldsymbol{a} = (a_1, a_2, a_3) = (1, 2, 3)$. The basis and its m-dual are

$$\boldsymbol{V} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 8 & 0 \\ 0 & 0 & 8 \end{pmatrix} \qquad \text{and} \qquad \boldsymbol{W} = \begin{pmatrix} 8 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & 0 & 1 \end{pmatrix}.$$

Both the primal and the m-dual are integral lattices that contain $8\,\mathbb{Z}^3$. If we take the projection of this lattice over the coordinates $I = \{2, 3\}$, we have a set of generating vectors given by the three rows of the last two columns of $\boldsymbol{V}$. When we upper-triangulate this 3×2 matrix, we obtain $c_1 = \gcd(2, 8) = 2$. The triangular basis matrix $\boldsymbol{V}'$ of this projection and its m-dual basis $\boldsymbol{W}'$ are then

$$\boldsymbol{V}' = \begin{pmatrix} 2 & 3 \\ 0 & 4 \end{pmatrix} \qquad \boldsymbol{W}' = \begin{pmatrix} 4 & 0 \\ -3 & 2 \end{pmatrix}.$$

A shortest vector in this m-dual basis is $(1, 2)$, whose Euclidean length is $\sqrt{5}$. The distance between successive lines that cover all the points in the non-rescaled version L_I of the primal projection is $1/\sqrt{5}$. This is the relevant quantity to measure the uniformity of the projection. If we project directly the initial m-dual basis over the coordinates in $I = \{2, 3\}$, we obtain a lattice whose basis is the identity, which is quite different than the $\boldsymbol{W}'$ above. A shortest nonzero vector in the m-dual has length 1 in that case.

There are similar construction procedures for the lattices generated by an MRG; this is explained in the LatMRG guide [46].

6 Computing the m-Dual of a Basis

The general case. For a given (integer) square basis matrix $\boldsymbol{V}$, one can obtain the corresponding (integer) m-dual basis $\boldsymbol{W}$ by solving the linear system $\boldsymbol{V}\boldsymbol{W}^{\mathsf{t}} = m\boldsymbol{I}$, in which all matrix entries are integer. In other words, $\boldsymbol{W}^{\mathsf{t}} = m\boldsymbol{V}^{-1}$. It can be computed exactly (in integer arithmetic) as follows. Let $\mathrm{adj}(\boldsymbol{V})$ denote the adjugate (or adjoint) of $\boldsymbol{V}$, which is the transpose of the cofactor matrix $\boldsymbol{C}$, whose element (i, j) is $c_{i,j} = (-1)^{i+j} d_{i,j}$, where $d_{i,j}$ is the determinant of the $(t-1) \times (t-1)$ matrix obtained by deleting row i and column j of $\boldsymbol{V}$. One has

$$\boldsymbol{V} \cdot \mathrm{adj}(\boldsymbol{V}) = \mathrm{adj}(\boldsymbol{V}) \cdot \boldsymbol{V} = \det(\boldsymbol{V}) \cdot \boldsymbol{I},$$

which is a diagonal matrix whose diagonal elements are all equal to $\det(\boldsymbol{V})$. In our case, all the $d_{i,j}$ are integers, so all elements of $\mathrm{adj}(\boldsymbol{V})$ are integers. Clearly, we have $\boldsymbol{W}^{\mathsf{t}} = (m/\det(\boldsymbol{V}))\mathrm{adj}(\boldsymbol{V})$, or equivalently $\boldsymbol{W} = (m/\det(\boldsymbol{V}))\boldsymbol{C}$. Since all elements of $\boldsymbol{W}$ are known to be integers, all the elements of $\mathrm{adj}(\boldsymbol{V})$ must be divisible by $\det(\boldsymbol{V})/m$, which is also an integer. We have an implementation of this only for integers in `ZZ` representation. It uses the NTL function `inv` from `mat_ZZ.h` to compute $\boldsymbol{C}$ in integer arithmetic, then it divides its elements by $\det(\boldsymbol{V})/m$. This is implemented by `mDualBasis` in `BasisConstruction.h`.

The two triangular methods that follow are much faster than this general method. See the speed comparisons in the `TestBasisConstructSpeedTri` example of Section 12.5.

The upper-triangular case. If $\boldsymbol{V}$ is upper-triangular, which occurs for example if it was constructed from a set of generating vectors by the gcd method described in Subsection 5, this system can be solved easily to yield a *lower-triangular* $\boldsymbol{W}$, with explicit formulas. If $v_{i,j}$ and $w_{i,j}$ are the elements of $\boldsymbol{V}$ and $\boldsymbol{W}$, respectively, then we can write (with $\boldsymbol{W}$ transposed)

$$\begin{pmatrix} v_{1,1} & \dots & v_{1,t} \\ \vdots & \ddots & \vdots \\ 0 & \dots & v_{t,t} \end{pmatrix} \begin{pmatrix} w_{1,1} & \dots & 0 \\ \vdots & \ddots & \vdots \\ w_{t,1} & \dots & w_{t,t} \end{pmatrix} = \boldsymbol{V}\boldsymbol{W}^{\mathrm{t}} = m\boldsymbol{I},$$

whose solution is given by $w_{i,i} = m/v_{i,i}$ for $i = 1, \dots, t$ and

$$w_{i,j} = -\frac{1}{v_{j,j}} \sum_{k=j+1}^{i} v_{k,j} w_{i,k}$$

for $1 \le j < i \le t$ [75]. All these $w_{i,j}$'s are integers, as are all the terms of the sum and also $v_{j,j}$, so all the computations can be made exactly in integer arithmetic with ordinary integer division, by using large enough integers. This is implemented by `mDualUpperTriangular` in `BasisConstruction.h`.

The lower-triangular case. Essentially the same algorithm works for a lower-triangular basis $\boldsymbol{V}$. We can write

$$\begin{pmatrix} v_{1,1} & \dots & 0 \\ \vdots & \ddots & \vdots \\ v_{t,1} & \dots & v_{t,t} \end{pmatrix} \begin{pmatrix} w_{1,1} & \dots & w_{1,t} \\ \vdots & \ddots & \vdots \\ 0 & \dots & w_{t,t} \end{pmatrix} = \boldsymbol{V}\boldsymbol{W}^{\mathrm{t}} = m\boldsymbol{I},$$

whose solution is $w_{i,i} = m/v_{i,i}$ for $i = 1, \dots, t$ and

$$w_{i,j} = -\frac{1}{v_{j,j}} \sum_{k=i}^{j-1} v_{j,k} w_{i,k}$$

for $1 \le i < j \le t$. This is implemented by `mDualLowerTriangular`.

The m-dual of a projection. It is important to understand that in general, the m-dual of the projection Λ_I of a rescaled lattice Λ_t over a subset I of coordinates is not the same as the projection of the m-dual lattice $\Lambda_t^* = L_t^*$ over the same subset I of coordinates, even if no projection is losing points. The next example illustrates this.

Example 2. Take $m = 5$, $t = 2$, $I = \{1\}$, $\boldsymbol{v}_1 = (1, 2)$, and $\boldsymbol{v}_2 = (0, 5)$. The m-dual basis in this case is $\boldsymbol{w}_1 = (5, 0)$ and $\boldsymbol{w}_2 = (-2, 1)$, and the m-dual lattice Λ_t^* turns out to be the same as Λ_t. We have

$$\boldsymbol{V} = \begin{pmatrix} 1 & 2 \\ 0 & 5 \end{pmatrix} \quad \text{and} \quad \boldsymbol{W} = \begin{pmatrix} 5 & 0 \\ -2 & 1 \end{pmatrix}.$$

This lattice and its m-dual both have density $1/5$ in two dimensions. The projection Λ_I of Λ_t over any single coordinate is $\mathbb{Z}$, the set of all integers, which has density 1, and the same holds for the projections of the m-dual Λ_t^*. The points never project onto each other. The m-dual of the projection $\Lambda_I = \mathbb{Z}$ is $\Lambda_I^* = m\mathbb{Z} = \{\ldots, -10, -5, 0, 5, 10, \ldots\}$, which has density $1/5$. This is also the dual of $L_I = \mathbb{Z}/5 = \{\ldots, -2/5, -1/5, 0, 1/5, 2/5, \ldots\}$, in which the distance between "hyperplanes" is $1/5$, the inverse of the length of the shortest nonzero vector in $m\mathbb{Z}$. This m-dual $m\mathbb{Z}$ differs from the projection of Λ_t^* over a single coordinate, which is $\mathbb{Z}$. So the projection of the m-dual is not the same as the m-dual of the projection. This occurs because the density of the projection Λ_I is not the same as the density of the full lattice Λ_t.

7 Lattice Basis Reduction

In Euclidean spaces, which are closed under linear combinations with real coefficients, it is trivial to select an orthogonal basis, i.e., a finite set of vectors which are pairwise-orthogonal and generate the space, and these vectors can be as small as we want. For lattices over $\mathbb{Z}$, which are closed to linear combinations with integer coefficients only, things are more complicated. In this setting, it is often of interest to find a lattice basis which is as orthogonal as possible and whose vectors are as short as possible, in some sense. This idea of reduced basis has many specific definitions. We recall and explain those that are implemented in Lattice Tester. In particular, basis reduction is an essential first step when we want to find a shortest nonzero vector in the lattice (the SVP problem). In Sections 7 to 10, unless indicated otherwise, the vector lengths are Euclidean lengths (i.e., we assume that the L^2 norm is used), and what we say applies to a lattice L_t which is not necessarily rescaled to integers. In the software, on the other hand, the lattices are always rescaled to integers, to make sure that the basis vectors can always be represented exactly.

7.1 Basis reduction in one dimension

The simplest case is that of a one-dimensional lattice. Then, a basis is just a single nonzero integer $\boldsymbol{v}_1 = c \neq 0$ and the lattice is the set of all (integer) multiples of c. The two vectors c and $-c$ are the shortest nonzero vectors. Moreover, as we saw earlier, the lattice generated by two or more distinct integers has a basis (and shortest vector) c given by the gcd of these integers.

7.2 Basis reduction in two dimensions

For a two-dimensional lattice ($t = 2$), one can reduce a given basis $\{\boldsymbol{v}_1, \boldsymbol{v}_2\}$ using an extension of Euclid's algorithm, called the *Lagrange reduction* method [26, 69], defined as follows. The method is often attributed to Gauss and named *Gaussian reduction* or Gauss-Lagrange reduction [5].

We start with the two basis vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ sorted by increasing (Euclidean) length, so $\boldsymbol{v}_1$ is the shortest. At each step, we try to reduce $\boldsymbol{v}_2$ as much as possible by subtracting an integer multiple of $\boldsymbol{v}_1$. The optimal multiple to subtract turns out to be

$$q = \mathsf{round}\,(\boldsymbol{v}_1 \cdot \boldsymbol{v}_2 / \boldsymbol{v}_1 \cdot \boldsymbol{v}_1),$$

where round means rounding to the nearest integer and we always break equalities by taking the value closest to 0 (e.g., $\mathsf{round}\,(\pm 1/2) = 0$). If $q = 0$, we stop, otherwise $\boldsymbol{v}_2$ is now shorter than $\boldsymbol{v}_1$, so we swap the two vectors and try again to reduce $\boldsymbol{v}_2$ with $\boldsymbol{v}_1$. At the end of the procedure, $\boldsymbol{v}_1$ is a shortest nonzero vector in the lattice and there exists no other basis whose two vectors are both smaller than $\boldsymbol{v}_2$. This basis is reduced in the strongest possible way, in all the senses that we will discuss later.

7.3 Successive minima

The *successive minima* of a lattice L_t are the positive real numbers $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_t < \infty$ for which λ_j is the smallest real number λ for which L_t contains j linearly independent vectors of L_t whose length does not exceed λ. A basis of t vectors for which vector j has length λ_j cannot be reduced further. It is important to realize, on the other hand, that in four or more dimensions, a set of linearly independent vectors that satisfy this condition is not necessarily a basis. And in five or more dimensions, there may exist no lattice basis $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ for which $\|\boldsymbol{v}_j\| = \lambda_j$ for all j. See [69, Page 33].

The convex body theorem of Minkowski states that if C is a convex subset of $\mathbb{R}^t$ symmetric about the origin, and if $\mathrm{vol}(C) \geq 2^t \det(L_t)$, then C contains at least one nonzero vector of L_t. As a special case, by taking C as a t-dimensional ball of radius r, $C = \{\boldsymbol{v} \in \mathbb{R}^t \text{ such that } \|\boldsymbol{v}\| \leq r\}$, we have $\mathrm{vol}(C) = r^t V_t$ where V_t is the volume of a unit ball in t dimensions, and C must contain a nonzero lattice point as soon as $r^t V_t \geq 2^t \det(L_t)$, i.e., when $r \geq 2[\det(L_t)/V_t]^{1/t}$. This gives an upper bound on the length of the shortest nonzero vector in L_t:

$$\lambda_1 \leq 2[\det(L_t)/V_t]^{1/t}. \tag{4}$$

Rogers [73] proved that the 2 in this inequality can be replaced by $\sqrt{t}/3$. This gives the upper bound

$$\lambda_1 \leq \min(2, \sqrt{t}/3)[\det(L_t)/V_t]^{1/t}. \tag{5}$$

In [74], he proved even better bounds, given by asymptotic expressions; see Section 9. The θ_n given in the last equation on page 39 of [74] is an upper bound on $\mathrm{vol}(C)/\det(L_t)$. For the Euclidean norm, this gives $r^t V_t \geq \theta_t \det(L_t)$ and then the upper bound $\lambda_1 \leq [\theta_t \det(L_t)/V_t]^{1/t}$.

A second theorem of Minkowski states that

$$\frac{2^t}{t!} \leq \frac{V_t}{\det(L_t)} \prod_{j=1}^{t} \lambda_j \leq 2^t. \tag{6}$$

These results are valid not only for the Euclidean norm, but for other norms as well. For the L^1 norm, for example, we have $V_t = 2^t$, so the upper bound in (4) becomes $\lambda_1 \leq [\det(L_t)]^{1/t}$.

In one and two dimensions, the simple gcd-based methods mentioned in Sections 7.1 and 7.2 easily provide a basis whose vector lengths reach the successive minima. But in more than two dimensions, there is no such effective method that reduces the basis vectors in an optimal way. There are also different definitions of the notion of *reduced basis*. We discuss some of them.

7.4 Gram-Schmidt orthogonalization and size-reduced basis

For any set of linearly independent vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_d$ in $\mathbb{R}^t$, with $d \leq t$, the *Gram-Schmidt orthogonalization (GSO)* process produces a set of d orthogonal vectors $\boldsymbol{v}_1^*, \ldots, \boldsymbol{v}_d^*$ as follows. Let $\boldsymbol{v}_1^* = \boldsymbol{v}_1$ and for $i = 2, \ldots, d$,

$$\boldsymbol{v}_i^* = \boldsymbol{v}_i - \sum_{j=1}^{i-1} \mu_{i,j} \boldsymbol{v}_j^*$$

where

$$\mu_{i,j} = \frac{\boldsymbol{v}_i \cdot \boldsymbol{v}_j^*}{\boldsymbol{v}_j^* \cdot \boldsymbol{v}_j^*}.$$

In the real space, these these vectors $\boldsymbol{v}_i^*$ are exactly orthogonal and form a basis. The basis vectors can also be made as small as we want. But for a lattice basis, this does not hold. The coefficients $\mu_{i,j}$ are generally not integers, so even if the $\boldsymbol{v}_i$ are lattice vectors, the $\boldsymbol{v}_i^*$ are generally not. In lattice reduction, the aim to obtain a basis formed by lattice vectors $\boldsymbol{v}_i$ that are "close" to the $\boldsymbol{v}_i^*$ in some sense, and nearly orthogonal.

The reduced vector $\boldsymbol{v}_i^* \in \mathbb{R}^t$ is the projection of $\boldsymbol{v}_i$ over the subspace that is orthogonal to the space generated by $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{i-1}$. If $\boldsymbol{v}_i$ is nearly orthogonal to that space, then $\mu_{i,1}, \ldots, \mu_{i,i-1}$ will all be close to 0. A basis $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ is said to be *size-reduced* if for all $1 \leq j < i \leq t$, $|\mu_{i,j}| \leq 1/2$. This means that one cannot reduce the length of the vector $\boldsymbol{v}_i$ by subtracting from it an integer multiple of $\boldsymbol{v}_j$ for some $j < i$. The LLL algorithm discussed in Section 7.8 provides such a size-reduced basis.

7.5 Korkine-Zolotarev-reduced basis

A basis $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ is called *Korkine-Zolotarev-reduced* (KZ-reduced, also called Hermite-reduced or Hermite-Korkine-Zolotarev-reduced by some authors [5]) if (a) it is size-reduced and (2) for $1 \leq i \leq t$, $\boldsymbol{v}_i^*$ has the same length as a shortest nonzero vector in the projection of L_t on the subspace that is orthogonal to the space generated by $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{i-1}$.

An equivalent (recursive) definition is that (i) $|\mu_{i,1}| \leq 1/2$ for $1 < i \leq t$; (ii) $\boldsymbol{v}_1$ is a shortest nonzero vector in L_t; and (ii) the projections $\boldsymbol{v}_i - \mu_{i,1}\boldsymbol{v}_1$ of $\boldsymbol{v}_2, \ldots, \boldsymbol{v}_t$ on the subspace that is orthogonal to $\boldsymbol{v}_1$ is Korkine-Zolotarev-reduced.

Algorithms to compute a KZ-reduced basis are given in [19, 23, 24] and further explained in [5, Chapter 11]. These algorithms are very costly in CPU time; they running time is exponential in t. Computing a KZ-reduced basis is obviously more expensive than finding a shortest nonzero lattice vector. In practice, we use a weakened version of KZ reduction named BKZ, described in Section 7.9.

7.6 Minkowski-reduced basis

A basis $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ is *Minkowski-reduced* if for each $i < t$, given $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{i-1}$, we have that $\boldsymbol{v}_i$ is a shortest nonzero lattice vector for which $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_i$ can be extended to a basis of L_t.

The ratio of lengths of the shortest and longest vectors in a Minkowski-reduced basis, $\|\boldsymbol{v}_1\|/\|\boldsymbol{v}_t\|$, is called the *Beyer quotient* [3]. In general, a given lattice may have several Minkowski-reduced bases. All of them must have the same length for $\boldsymbol{v}_1$ (a shortest vector), but the vectors $\boldsymbol{v}_i$ for $i > 6$ may have different lengths. In particular, two Minkowski-reduced bases may have different lengths for their longest vector $\boldsymbol{v}_t$, which means that for $t > 6$, the Beyer quotient of a lattice L_t is not unique. To make it unique, one could take the shortest $\boldsymbol{v}_t$ over all Minkowski-reduced bases, i.e., define the Beyer quotient of L_t as the largest ratio $\|\boldsymbol{v}_1\|/\|\boldsymbol{v}_t\|$ over all Minkowski-reduced bases.

Algorithms to compute a Minkowski-reduced basis can be found in [1, 2, 16, 19]. These algorithms take exponential time in t. The algorithm of [1], implemented in Lattice Tester, makes successive applications of a procedure that solves the SVP, with some additional constraints. In t dimension, at least t SVPs must be solved (and sometimes much more), which means that computing a Minkowski-reduced basis is generally much more time-consuming than computing the shortest vector.

Dealing with the round-off errors during the Cholesky decomposition when solving the SVP (see Section 8) is an important question here, because a small error in the bounds of the BB algorithm can lead to wrong results. One may miss a shorter vector and, as a result, (conceivably) not obtain a true Minkowski-reduced basis at the end of the reduction algorithm. In that case, one may consider redoing the computations with arbitrary-precision floating-point numbers, for verification. This is of course much slower.

7.7 Pairwise reductions

There is no easy and effective generalization of Euclid's algorithm to compute a KZ-reduced or Minkowski-reduced basis in more than two dimensions. Known algorithms have computing times that grow exponentially in the dimension. For this reason, various heuristics have been proposed to reduce the length of basis vectors at low cost.

One simple heuristic discussed and recommended in [13, 25] is a *pairwise reduction* method, which consists in trying to reduce the length of a basis vector $\boldsymbol{v}_i$ by subtracting from it q times another basis vector $\boldsymbol{v}_j$, for some integer q and given indices $i \neq j$. The Euclidean length of the new vector $\boldsymbol{v} = \boldsymbol{v}_i - q\boldsymbol{v}_j$ is minimized by taking $q = \mathsf{round}(\boldsymbol{v}_i \cdot \boldsymbol{v}_j / \boldsymbol{v}_j \cdot \boldsymbol{v}_j)$, where $\mathsf{round}(x)$ denotes the integer nearest to x. If this q is nonzero, we can replace $\boldsymbol{v}_i$ by the strictly shorter vector $\boldsymbol{v}_i - q\boldsymbol{v}_j$ in the basis. This can be tried with all pairs $i \neq j$, and repeated until running through all pairs gives no further improvement. See [75, Algorithm 2.4.1].

For a two-dimensional lattice, this is equivalent to the Lagrange reduction method discussed earlier, which always provides a shortest lattice vector. In more than two dimensions, this is only a heuristic and does not provide a shortest vector in general.

When $\boldsymbol{v}_i$ is replaced by $\boldsymbol{v}_i - q\boldsymbol{v}_j$ in the basis, to update the dual basis at the same time, it suffices to replace $\boldsymbol{w}_j$ by $\boldsymbol{w}_j + q\boldsymbol{w}_i$. One can easily verify that this preserves the duality property. More generally, if $\boldsymbol{v}_i$ is replaced by $\boldsymbol{v}_i - \sum_{j \neq i} q_j \boldsymbol{v}_j$ for some integers q_j, the dual basis can be updated by replacing $\boldsymbol{w}_j$ by $\boldsymbol{w}_j + q_j \boldsymbol{w}_i$ for each $j \neq i$.

Dieter [13] suggested to also apply pairwise reductions to the dual basis. The motivation was that reducing the vectors of the dual basis often lead to a reduction of the vectors of the primal basis. To do this, we try to reduce the length of a dual basis vector $\boldsymbol{w}_i$ by replacing this vector by $\boldsymbol{w}_i - q\boldsymbol{w}_j$, where $q = \mathsf{round}(\boldsymbol{w}_i \cdot \boldsymbol{w}_j / \boldsymbol{w}_j \cdot \boldsymbol{w}_j)$. If $q \neq 0$ and if the corresponding primal basis vector $\boldsymbol{v}_j + q\boldsymbol{v}_i$ is not longer than $\boldsymbol{v}_j$, then we make the replacement. This can also be tried for all pairs $i \neq j$, and repeated until it gives no improvement. However, these reductions in the dual basis are not really useful, because the pairwise reduction with q in the dual is exactly equivalent to a pairwise reduction in the primal basis with $-q$ instead of q. Moreover, the LLL and BKZ reductions described in the next subsections are much more effective than these pairwise reductions, and adding the pairwise reductions either before or after them does not really help. Therefore, we recommend to ignore those pairwise reductions. On the other hand, LLL and BKZ are actually comprised of a succession or combination of several pairwise reductions.

7.8 LLL reduction

In a landmark paper, Lenstra, Lenstra, and Lovász [64] proposed a form of lattice basis reduction known as *LLL reduction*, which in a sense generalizes the Lagrange reduction, and became very important and influential. The book [70] covers much of the developments of LLL reduction and its variants in the 30 years that followed. For $1/4 < \delta \leq 1$, we say that a basis is *LLL reduced with factor δ* if

1. it is sized-reduced;
2. for $1 \leq i < t$, we have $\delta \, \|\boldsymbol{v}_i^*\|^2 \leq \|\boldsymbol{v}_{i+1}(i)\|^2 = \|\boldsymbol{v}_{i+1}^* + \mu_{i+1,i}\boldsymbol{v}_i^*\|^2$, where $\boldsymbol{v}_j(i)$ denotes the component of $\boldsymbol{v}_j$ which is orthogonal to $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{i-1}$.

For such a reduced basis, we have the following bounds on the squared ratio between the length of $\boldsymbol{v}_i$ and the ith successive minimum λ_i, for $i = 1, \ldots, t$ [64]:

$$(\delta - 1/4)^{i-1} \leq \left(\frac{\|\boldsymbol{v}_i\|}{\lambda_i}\right)^2 \leq \left(\frac{1}{\delta - 1/4}\right)^{t-1}. \tag{7}$$

In practice, the ratio is often equal (or very close) to 1 when i is not too large, e.g., less than 20. Then, the length of $\boldsymbol{v}_1$ is at most slightly larger than the length d_t of a shortest nonzero vector in the lattice, and the lengths of the other basis vectors are also not much larger than the shortest possible. We will illustrate this later with numerical examples. For this reason, an LLL-reduced basis tends to yield a much thinner BB tree when computing the shortest nonzero vector as in Section 8. Some authors fix δ at 3/4 [19, 65, 70], but we prefer values closer to 1. In Lattice Tester, we use $\delta = 0.99999$ as the default value for both LLL and BKZ in `ReducerStatic.h`, but the user can pass another value as a parameter. In NTL, the recommended and default value is 0.99.

Note that the vectors of a LLL-reduced basis are not necessarily sorted by increasing order of length, and $\boldsymbol{v}_1$ is not necessarily the shortest vector in the basis, but when δ is close to 1, it is typically not much longer than the shortest vector in the basis or in the lattice. The right side of (7) for $i = 1$ and $\delta \approx 1$ gives approximately $\|\boldsymbol{v}_i\|/\lambda_i \leq (4/3)^{(t-1)/2}$.

LLL reduction is weaker than other forms such as KZ or Minkowski reductions. In particular, it does not guarantee that the shortest basis vector $\boldsymbol{v}_1$ is a shortest nonzero lattice vector. However, in contrast to these other reduction forms, LLL-reduction only takes polynomial time. A key idea to obtain a proven polynomial-time bound was to ignore swaps (reorderings) of basis vectors that bring only a marginal gain. This controls the total number of swaps and then the total work. If we do all the swaps that bring some gain (i.e., take $\delta = 1$) there is no proof of polynomial time, although the method still work well in practice in most cases. See [69].

The original LLL algorithm is stated in [64, 77] and [75, Algorithm 2.4.5]. It uses at most $\mathcal{O}(t^4 \log \|\boldsymbol{v}_t\|^2)$ operations where $\boldsymbol{v}_t$ is the longest vector in the original basis. Storjohann [82] proposed a modified LLL algorithm that requires at most $\mathcal{O}(t^3 \ln \|\boldsymbol{v}_t\|^2)$ operations instead. It is used in FLINT; see sections 22.31 to 22.33 of [18]. He also gave an algorithm that achieves a better rate using fast matrix multiplication techniques. However, the latter is likely to be faster only for very large t, due to a larger hidden constant. See [68, 78] for surveys of other variants and improvements and their complexity analysis.

In Lattice Tester, we use the efficient C++ implementations of the original LLL available in NTL [79], with very small changes for example to recover the square lengths of the basis vectors. The NTL implementations are available for all the different types of `Real`, but the basis vector coordinates must be represented in the type `NTL::ZZ`, so they are stored exactly as integers of arbitrary length. In these implementations, the (squared) vector lengths and the $\mu_{i,j}$ can be represented either approximately in floating point or exactly as large integers or as rational numbers (ratios of two integers). In the latter case, all the computations are exact but take more time. The floating-point versions are faster and usually sufficient when using LLL for pre-reduction. The floating-point numbers can be either in double precision

(64 bits), quadruple precision (128 bits), or with arbitrary precision (the RR type offered by NTL). The input vectors do not have to be independent; they can be just a set of dependent generating vectors, and the functions will return an LLL-reduced basis preceded by some zero vectors in the output matrix.

We also made an implementation for the types `<Int = int64_t, Real = double>`, but it is not really faster than the NTL implementation with `<Int = NTL::ZZ, Real = double>`.

7.9 Schnorr's blockwise Korkine-Zolotarev (BKZ) reduction

The concept of BKZ reduction was introduced by Schnorr [76] and a practical algorithm to obtain a BKZ-reduced basis is given in [77]. There are also NTL implementations (exact or floating-point) which are available in Lattice Tester, just like for LLL. BKZ generalizes LLL in the sense that instead of imposing only the restriction that $\delta\|\boldsymbol{v}_i^*\|^2 \leq \|\boldsymbol{v}_{i+1}(i)\|^2$ for each i, this condition is extended to the $k-1$ vectors $\boldsymbol{v}_{i+1}(i), \ldots, \boldsymbol{v}_{\min(i+k-1,t)}(i)$. For $1/4 < \delta < 1$, a basis is said to be *BKZ-reduced for block sizes k with factor δ* if

(i) it is sized-reduced;

(ii) for each $i = 1, \ldots, t-k+1$, the k vectors $\boldsymbol{v}_i(i), \ldots, \boldsymbol{v}_{\min(i+k-1,t)}(i)$ are Korkine-Zolotarev-reduced with a factor δ, which is equivalent to saying that

$$\delta\|\boldsymbol{v}_i^*\|^2 \leq [\lambda_1(O_i(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{\min(i+k-1,t)}))]^2,$$

where $\lambda_1(O_i(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_j))$ denotes the length of a shortest vector in the lattice $O_i(\boldsymbol{v}_1, \ldots, \boldsymbol{v}_j)$ which is the projection of the lattice generated by $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_j$ over the space that is orthogonal to that spanned by $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_{i-1}$.

With $k = 2$, this is equivalent to LLL-reduction with factor δ.

The shortest vector $\boldsymbol{v}_1$ in such a reduced basis satisfies

$$\|\boldsymbol{v}_1\|^2/\lambda_1^2 \leq \alpha_k^{(t-1)/(k-1)}$$

where $\alpha_k \leq k^{1+\ln k}$, when $k-1$ divides $t-1$ [76]. We recommend BKZ as the most effective reduction method in practice. We will see in the numerical examples that in large dimensions, applying only LLL to reduce the basis before computing a shortest vector is often insufficient and leads to failure, whereas BKZ with a block size of 10 or 12 is much more effective.

8 Shortest Vector Problem

8.1 Problem formulation

In this section, we assume that we have a integral lattice, so all basis coordinates are integers. Computing the exact length of a *shortest nonzero vector in* Λ_t for large t is known to be a very hard problem. This length is the first minimum, λ_1, but we also denote it by d_t to make the dependence on t more explicit. (Often, we want to compute it for several projections of the original lattice over subsets of $s < t$ coordinates.) In Lattice Tester, we compute d_t with a BB procedure adapted from [14]. This exact procedure can be very costly (exponential in t in the worst case). It is generally much faster when the lattice basis has been pre-reduced using some of the procedures described in Section 7, most notably LLL and BKZ. After these pre-reductions, the basis vectors are typically shorter and the length of the shortest basis vector is not much larger than d_t. In fact, it is often equal to d_t, but we do not know for sure that there is no shorter vector until we have applied the full BB procedure. When t is large, LLL and BKZ rarely yield a shortest vector. See the numerical examples in Sections 12.4 and 12.8. On the other hand, the length of the current shortest basis vector always provides an upper bound on d_t. When making a computer search for good lattices, if the current shortest vector is deemed too small, we can immediately reject the current candidate and stop the computations for this candidate, to save time.

Let $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ be independent vectors that form a lattice basis, and are the rows of $\boldsymbol{V}$. Computing a shortest non-zero vector amounts to finding integers $z_1, \ldots, z_t$, not all zero, such that the vector $\boldsymbol{v} = (v_1, \ldots, v_t) = z_1\boldsymbol{v}_1 + \cdots + z_t\boldsymbol{v}_t = \boldsymbol{z}\boldsymbol{V}$ is as short as possible. Trying all combinations for those z_j's is not a viable option. Suppose we want to find the shortest vector with respect to the L^p norm, defined by $\|\boldsymbol{v}\|_p = (|v_1|^p + \cdots + |v_t|^p)^{1/p}$. We can formulate this as an integer programming optimization problem with decision variables $z_1, \ldots, z_t$, as follows:

$$\text{Minimize} \qquad \|\boldsymbol{v}\|_p \tag{8}$$

$$\text{Subject to} \qquad \boldsymbol{v} = \boldsymbol{z}\boldsymbol{V} = \sum_{i=1}^{t} z_i\boldsymbol{v}_i, \qquad z_i \in \mathbb{Z}, \qquad \sum_{i=1}^{t} |z_i| > 0. \tag{9}$$

An optimal solution to this problem with $p = 2$ is a vector of length d_t. We denote by $b(p)$ the L^p norm of the shortest currently-known vector in this lattice. This $b(p)$ is an upper bound on the length of a shortest vector, and when running the algorithm, we are only interested in finding vectors shorter than that.

In our BB algorithm, we fix successively z_t, then z_{t-1}, then z_{t-2}, etc. At each step, for any fixed values of $z_t, \ldots, z_{j+1}$ that we consider, we compute a finite range (interval) of values of z_j such that all values outside this range cannot lead to a better solution, and we scan the values of z_j in this range. A key part of the algorithm is how we obtain the bounds that define this range. In the following two subsections, we explain two ways of doing that: (1) with a triangular basis and (2) with a Cholesky decomposition of the matrix of scalar products of basis vectors. The set of all partial solutions form a tree called the BB tree.

To compute a shortest vector in the m-dual lattice, it suffices to interchange the primal and m-dual bases (dualize), and everything else works the same.

8.2 Bounds obtained using a lower-triangular basis

A first approach assumes that $\boldsymbol{V} = \boldsymbol{L}$, a *lower-triangular matrix* with integer elements $\ell_{i,j}$ that can be represented exactly. This $\boldsymbol{L}$ may have been obtained via the gcd construction method of Section 5.2. The lattice can be Λ_t or its m-dual Λ_t^*, for example. Recall that any lattice vector $\boldsymbol{v}$ can be written as $\boldsymbol{v} = (v_1, \ldots, v_t) = \boldsymbol{z}\boldsymbol{L}$ for $\boldsymbol{z} = (z_1, \ldots, z_t)$ and that $\ell_{i,k} = 0$ for $i < k$, so we have

$$v_k = \sum_{i=k}^{t} z_i \ell_{i,k} = z_k \ell_{k,k} + r_k \qquad \text{where} \qquad r_k = \sum_{i=k+1}^{t} z_i \ell_{i,k}.$$

We can assume that $z_t \geq 0$, because if $z_t < 0$, we can simply replace $\boldsymbol{z}$ by $-\boldsymbol{z}$ and it gives a lattice vector of the same length. To save work, we never consider negative values for z_t. We denote the partial sum

$$s_j(p) = \sum_{k=j+1}^{t} |v_k|^p,$$

which depends on $z_t, \ldots, z_{j+1}$. Then, for any $1 \leq j \leq t$, we have

$$\|\boldsymbol{v}\|_p^p \;\geq\; \sum_{k=j}^{t} |v_k|^p \;=\; s_{j-1}(p) \;=\; |v_j|^p + s_j(p) \;=\; |z_j \ell_{j,j} + r_j|^p + s_j(p). \tag{10}$$

Recall that $b(p)$ is the L^p norm of the shortest vector that has been found so far. This short vector does not have to be in the triangular basis, it can be found separately. The vector $\boldsymbol{v}$ can be shorter only if it satisfies $s_{j-1}(p) < b(p)^p$, for all $j > 0$. That is, we must have $|z_j \ell_{j,j} + r_j|^p < b(p)^p - s_j(p) > 0$, which implies

$$\left\lceil \frac{-(b(p)^p - s_j(p))^{1/p} - r_j}{\ell_{j,j}} \right\rceil \;\leq z_j \leq\; \left\lfloor \frac{(b(p)^p - s_j(p))^{1/p} - r_j}{\ell_{j,j}} \right\rfloor. \tag{11}$$

This can be rewritten as

$$z_j^{\min} = z_j^{\min}(p) := \lceil c_j - \delta_j(p) \rceil \;\leq z_j \leq\; \lfloor c_j + \delta_j(p) \rfloor =: z_j^{\max}(p) = z_j^{\max}, \tag{12}$$

where

$$c_j \;=\; -\frac{r_j}{\ell_{j,j}} = -\frac{1}{\ell_{j,j}} \sum_{i=j+1}^{t} z_i \ell_{i,j} = -\sum_{i=j+1}^{t} z_i \tilde{\ell}_{i,j} \tag{13}$$

is the center of the interval and does not depend on p, while

$$\delta_j(p) \;=\; \frac{(b(p)^p - s_j(p))^{1/p}}{\ell_{j,j}} \tag{14}$$

is the radius of the interval when $b(p)^p - s_j(p) > 0$ (otherwise the interval is empty). For $j = t$, we have $c_t = r_t = s_t(p) = 0$ and z_t is never negative, so the bounds on z_t simplify to

$$z_t^{\min} := 0 \ \le z_t \le \ \lfloor b(p)/\ell_{t,t} \rfloor =: z_t^{\max}. \tag{15}$$

For $j < t$, c_j and $\delta_j(p)$ depend on $z_t, z_{t-1}, \ldots, z_{j+1}$. What we have just done is to fix $z_t, \ldots, z_{j+1}$ and relax the integrality constraints on $z_j, \ldots, z_1$. When considering the possible values of z_j, we can restrict ourselves to the integers that lie in the interval specified by (12) and (15). The $\tilde{\ell}_{i,j} = \ell_{i,j}/\ell_{j,j}$ can be precomputed to avoid the divisions by $\ell_{j,j}$.

Add ϵ_1 and ϵ_2.

In view of (14) we prefer to have $b(p)$ as small as possible and $\ell_{j,j} > 0$ as large as possible, to reduce the range of values of z_j that must be examined. But as we have seen earlier, we always have $\prod_{j=1}^{t} \ell_{j,j} = \det(\Lambda_t)$, which is 1 over the density of the lattice, so it is not possible to increase all the $\ell_{j,j}$'s. Typically, the projection of the lattice L_t over any single coordinate contains all the integer multiples of $1/m$, so for Λ_t it contains all the integers. This implies in particular that $\ell_{1,1} = 1$ for an upper-triangular basis and $\ell_{t,t} = 1$ for any lower-triangular basis, so the range of values of z_t at the first level of the tree, given in (15), will usually be large unless $b(p)$ is already very small. This usually limits the efficiency of using a triangular basis for the BB. That is, we expect that too many nodes will have to be examined in the BB tree.

8.3 Bounds obtained via a Cholesky decomposition

Bounds for the L^2 norm. This approach works for the L^2 norm, i.e., for $p = 2$. As in [1, 14, 16, 71], for any given basis $\boldsymbol{V}$, we compute the Cholesky decomposition of the matrix $\boldsymbol{V}\boldsymbol{V}^{\mathrm{t}}$ of inner products of the basis vectors (also called the Gram matrix):

$$\boldsymbol{V}\boldsymbol{V}^{\mathrm{t}} = \boldsymbol{L}\boldsymbol{L}^{\mathrm{t}}$$

where $\boldsymbol{L}$ is a lower-triangular matrix with *real-valued* elements $\ell_{i,j}$. For any lattice vector $\boldsymbol{v}$, using the same notation as in (10), we can write

$$\|\boldsymbol{v}\|_2^2 \ = \ \boldsymbol{v}\boldsymbol{v}^{\mathrm{t}} = \boldsymbol{z}\boldsymbol{V}\boldsymbol{V}^{\mathrm{t}}\boldsymbol{z}^{\mathrm{t}} = \boldsymbol{z}\boldsymbol{L}\boldsymbol{L}^{\mathrm{t}}\boldsymbol{z}^{\mathrm{t}} = \sum_{k=1}^{t} v_k^2 \ \ge \ s_{j-1}(2) \ = \ |z_j\ell_{j,j} + r_j|^2 + s_j(2). \tag{16}$$

This gives the same bounds as in (11) and (12), valid only for $p = 2$, because for other values of p we do not have this quadratic form for the norm.

Some Cholesky decomposition algorithms return the upper triangular matrix $\boldsymbol{U} = \boldsymbol{L}^{\mathrm{t}}$ instead of $\boldsymbol{L}$. One can simply transpose it to get $\boldsymbol{L}$. Our implementation uses the LDL Cholesky decomposition [83] instead of the classical one. The LDL decomposition decomposes a symmetric positive definite matrix $\boldsymbol{A}$ (such as $\boldsymbol{A} = \boldsymbol{V}\boldsymbol{V}^{\mathrm{t}}$) as $\boldsymbol{A} = \tilde{\boldsymbol{L}}\boldsymbol{D}\tilde{\boldsymbol{L}}^{\mathrm{t}}$ where $\tilde{\boldsymbol{L}}$ is a lower-triangular matrix whose diagonal elements are all 1, and $\boldsymbol{D}$ is a diagonal matrix

whose elements are all positive (when $\boldsymbol{A}$ is positive definite). The advantage of this decomposition is that it does not require taking square roots. It works as follows, where the $a_{i,j}$, $\tilde{\ell}_{i,j}$, and d_j are the elements of $\boldsymbol{A}$, $\tilde{\boldsymbol{L}}$, and $\boldsymbol{D}$ (this d_j should not be confounded with the length d_t defined in Section 8.1):

$$\begin{aligned} d_1 &= a_{1,1}, \\ \tilde{\ell}_{2,1} &= a_{2,1}/d_1, \\ d_2 &= a_{2,2} - (\tilde{\ell}_{2,1})^2 d_1, \\ &\vdots \\ d_j &= a_{j,j} - \sum_{k=1}^{j-1} (\tilde{\ell}_{j,k})^2 d_k, \\ \tilde{\ell}_{i,j} &= \frac{1}{d_j}\left(a_{i,j} - \sum_{k=1}^{j-1} \tilde{\ell}_{i,k}\tilde{\ell}_{j,k} d_k\right) \qquad \text{for all } i > j. \end{aligned}$$

Note that $\tilde{\ell}_{j,j} = 1$. One has $\boldsymbol{L} = \tilde{\boldsymbol{L}}\boldsymbol{D}^{1/2}$ where $\boldsymbol{D}^{1/2}$ is a diagonal matrix with elements $d_j^{1/2}$. We can then recover $\ell_{i,j}$ via $\ell_{j,j} = d_j^{1/2}$ and $\ell_{i,j} = d_j^{1/2}\tilde{\ell}_{i,j}$ for all $i > j$. When computing c_j and $\delta_j(p)$ for the bounds (12) for $p = 2$, we can use the fact that $\ell_{i,j}/\ell_{j,j} = \tilde{\ell}_{i,j}$ and $\ell_{j,j}^p = d_j$, so there is no need to compute the $\ell_{i,j}$ explicitly. We can write (for $p = 2$):

$$\begin{aligned} c_j &= -\frac{r_j}{\ell_{j,j}} = \sum_{i=j+1}^{t} z_i \frac{\ell_{i,j}}{\ell_{j,j}} = \sum_{i=j+1}^{t} z_i \tilde{\ell}_{i,j} \qquad \text{and} \\ \delta_j(2) &= \frac{(b(2)^2 - s_j(2))^{1/2}}{\ell_{j,j}} \end{aligned}$$

where the $\ell_{i,j}$ and $s_j(2)$ come from the Cholesky decomposition, and we have the same formula as in (12) for the bounds:

$$\lceil c_j - \delta_j(2) \rceil \le z_j \le \lfloor c_j + \delta_j(2) \rfloor \tag{17}$$

for $j < t$ and $0 \le z_t \le \lfloor b(2)/\ell_{t,t} \rfloor$ as in (15). This is what we use in our implementation.

Bounds for the L^p norm for $0 < p < 2$ with the Cholesky decomposition. To compute the shortest nonzero vector with respect to the L^p norm instead of the Euclidean norm, Algorithm 3 with the Cholesky decomposition still works, with small modifications, when $0 < p < 2$. A better solution $\boldsymbol{v}$ must satisfy $\|\boldsymbol{v}\|_p < b(p)$. But since $\|\boldsymbol{v}\|_p \ge \|\boldsymbol{v}\|_2$ when $0 < p < 2$, a shorter $\boldsymbol{v}$ must satisfy $\|\boldsymbol{v}\|_2 \le \|\boldsymbol{v}\|_p < b(p)$, so we must have

$$b(p)^2 > \|\boldsymbol{v}\|_2^2 \ge \sum_{k=j}^{t} |v_k|^2 = |z_j \ell_{j,j} + r_j|^2 + s_j(2). \tag{18}$$

The same reasoning that led to (11) then gives

$$\lceil c_j - \delta_j(p) \rceil \ \leq z_j \leq \ \lfloor c_j + \delta_j(p) \rfloor \tag{19}$$

with

$$c_j = -\frac{r_j}{\ell_{j,j}} \qquad \text{and} \qquad \delta_j(p) = \frac{(b(p)^2 - s_j(2))^{1/2}}{\ell_{j,j}}$$

when we use the Cholesky decomposition to find a shortest vector for the L^p norm for $p < 2$. This is what we use in our implementation for $p = 1$.

The fact that $b(p)^2 \geq b(2)^2$ implies that the new bounds are wider, so the BB tree becomes larger and the algorithm takes more time to execute.

Another option would be to use the algorithm proposed by [13] and discussed in [25, Exercise 3.3.4-16]. This algorithm uses both the basis matrix $\boldsymbol{V}$ and its m-dual $\boldsymbol{W}$ to obtain different bounds that replace those in (11). We expect it to be slower.

8.4 Numerical errors in the decomposition and bounds

The LDL Cholesky factorization is computed in floating point, since the values are not necessarily integers, because of the divisions by d_j. There is a possibility that during the Cholesky decomposition, some d_j becomes negative for a large value of j, due to numerical round-off errors, particularly when m is large, and/or the dimension is large, or when the basis is not pre-reduced very well. When this happens, this means that the values we obtain make no sense and $\ell_{j,j}$ cannot be computed, so the BB algorithm fails when we use these bounds. The risk that this happens can be minimized by using the `RR` high-precision floating-point type from NTL (see Section 11.2) for the Cholesky factorization and for computing the bounds on the z_j. When the BB fails in this way with the `double` real-number representation, we can switch to `RR` and try again. We can also try to improve the pre-reduction by applying BKZ with a δ closer to 1 and perhaps a larger block size.

It is important to realize that even when all diagonal elements remain positive, there is still the possibility of an error in the bounds due to numerical imprecision, which can lead to an erroneous BB algorithm. The bounds in (12), (17), and (19) are computed in floating point before being truncated to integers. The computations are therefore not exact and there is a possibility of missing a valid value of z_j and then a slightly shorter vector because of that. If the error makes the bounds larger than they should be, we will visit more nodes than necessary in the BB algorithm, but this cannot lead to missing the shortest vector. If the error makes the bounds narrower than they should, then we could miss the shortest vector. In the software, one has the option to add a small safety margin $\epsilon > 0$ to the radius $\delta_j(2)$ in all the bounds (12), (17), and (19), before truncating them to integers.

[2] By default, $\epsilon = 0$, but this value can be increased (e.g., to $\epsilon = 10^{-6}$ or something like that) with the function `setEpsBounds` in the `ReducerBB` class. Note that if adding a very small safety margin ϵ leads to a new shortest vector, its length cannot be much smaller than the one obtained without the safety margin. Adding a larger ϵ will usually increase the running time without returning a much shorter vector. Adding this margin can help, but it does not solve all possible numerical issues. It does not prevent the numerical errors in the Cholesky decomposition. It is therefore recommended to check the calculations using `ZZ` + `RR` representations with larger precision for the RNGs that we want to retain, after making a search.

In an earlier version of our software, we were always maintaining the m-dual basis along with the primal basis and were using the dual basis to compute the second half of the Cholesky decomposition, for $j > t/2$, to improve stability. The computations were then always made in `double`. We no longer do that because we no longer maintain the dual basis. We rather use `RR` to obtain higher accuracy. The method of Section 8.2 avoids these numerical approximation issues but typically gives much wider bounds (some larger values of $\delta_j(p)$).

8.5 The branch-and-bound (BB) procedure

We now describe the BB procedure used to compute d_t by solving the integer programming problem (8–9). This algorithm works for all $p \geq 1$ if $\boldsymbol{L}$ is a triangular basis and for $0 < p \leq 2$ if $\boldsymbol{L}$ was obtained from a Cholesky decomposition. Recall that the bounds on z_t are $0 \leq z_t \leq \delta_t(p) = b(p)/\ell_{t,t}$. Here, we exploit the fact that if $\boldsymbol{v} = \boldsymbol{z}\boldsymbol{V}$ is a lattice vector, then $-\boldsymbol{v} = -\boldsymbol{z}\boldsymbol{V}$ is also a lattice vector with exactly the same length. This permits us to ignore the negative values of z_t and thereof cut the BB tree in half. We would like $b(p)$ to be as small as possible, to reduce the range of values that must be examined.

The algorithm explores a BB tree in which each node corresponds to a partial solution $(z_t, \ldots, z_{j+1})$. This node has a son $(z_t, \ldots, z_{j+1}, z_j)$ for each value of z_j that satisfies the bounds (12) and (15), or their equivalent when using Cholesky, and with ϵ possibly added as described in Section 8.4. The root corresponds to the initial state when no variable is fixed, the nodes at the first level is when only z_t is fixed, and so on. This root has a son for each z_t such that $0 \leq z_t \leq b(p)/\ell_{t,t}$. When no z_j satisfies the bounds (11), i.e., the interval is empty, the corresponding node has no son, so this branch is a dead end. When we reach a tree leaf that represents a full admissible solution $(z_t, \ldots, z_1)$, we have just found a new candidate vector $\boldsymbol{v}$ that may be shorter than our current shortest vector $\boldsymbol{v}_1$. If it is indeed shorter, we update $b(p)$, we take note of its length and of the corresponding $\boldsymbol{z}$, and we continue searching for a better $\boldsymbol{z}$ until we have explored the full tree. At the end, we return the best $\boldsymbol{z}$ that we

[2] From Pierre: We thought about adding the same ϵ to $b(p)^2 - s_j(2)$ when checking if this quantity is positive to see if there is still hope of finding a shorter vector in that branch before computing the bounds. But this sometimes leads to taking the square root of a negative value of $b(p)^2 - s_j(2)$ when computing '`dc`" in the recursive function `tryZShortVec`.

have found. In case there are many shortest vectors, we could easily also store and return all the corresponding $\boldsymbol{z}$'s.

Instead of always exploring the full tree with the same basis and same Cholesky decomposition as we do, another possibility could be that when we find a shorter vector $\boldsymbol{v}$, we update the basis so that the new basis will now contain $\boldsymbol{v}$ as its new $\boldsymbol{v}_1$, we change the other vectors (if needed) in a way that they still form a basis of our lattice, we redo the Cholesky decomposition, we exit completely from the recursive procedure and restart for $j = t$ with the new basis. This could be worthwhile if the length of $\boldsymbol{v}_1$ has reduced significantly, because the new tree is likely to be thinner, but not much if the initial $b(p)$ is very close to the length of a shortest vector, which usually happens when we use LLL and BKZ to pre-reduce beforehand. So we do not do this.

Our recursive implementation of this BB procedure is sketched in Algorithm 3. The initial call would be $\mathbf{BB}(\boldsymbol{V}, \boldsymbol{0}, (1, 0, \ldots, 0), t)$. At the end, $\boldsymbol{z}^*$ will contain an optimal $\boldsymbol{z}$. This procedure is implemented in `ReducerBB.h`.

Algorithm 3: : Recursive BB procedure for the SVP

$\mathbf{BB}(\boldsymbol{V}, \boldsymbol{z}, \boldsymbol{z}^*, j)$:
  // $z_{j+1}, \ldots, z_t$ are fixed and we examine the possible values for z_j;
  if $j = 0$ **then**
    // We are at a tree leaf;
    if $\boldsymbol{z} \neq 0$ and $\|\boldsymbol{z}\boldsymbol{V}\|_p < b(p)$ **then**
      $\boldsymbol{z}^* \leftarrow \boldsymbol{z}$ // We found a shorter nonzero vector;
    return;
  compute $s_j(p)$, r_j, and the bounds $z_j^{\min}$ and $z_j^{\max}$ in (12) or (15);
  if $z_j^{\min} > z_j^{\max}$ **then**
    return;
  // Otherwise, try all admissible values of z_j, starting from the center if $j < t$;
  if $j = t$ **then**
    $i_2 = 0$
  else
    $i_2 \leftarrow \lceil z_j^{\min} + z_j^{\max})/2 \rceil$;
  $i_1 \leftarrow i_2 - 1$;
  while $i_1 \geq z_j^{\min}$ or $i_2 \leq z_j^{\max}$ **do**
    if $i_1 \geq z_j^{\min}$ **then**
      $z_j \leftarrow i_1$; $\mathbf{BB}(\boldsymbol{V}, \boldsymbol{z}, \boldsymbol{z}^*, j-1)$; $i_1 \leftarrow i_1 - 1$;
    if $i_2 \leq z_j^{\max}$ **then**
      $z_j \leftarrow i_2$; $\mathbf{BB}(\boldsymbol{V}, \boldsymbol{z}, \boldsymbol{z}^*, j-1)$; $i_2 \leftarrow i_2 + 1$;
  return;

The total time taken by this BB algorithm is roughly proportional to the number of tree nodes that we visit, i.e., the number of calls to the recursive BB procedure. In the worst case, this number grows exponentially with the dimension. It is therefore important

to reduce the size of the tree as much possible. For this, it helps to start with $b(p)$ as small as possible, because this shortens the search interval determined by the bounds (11) at each level j, and can therefore greatly reduce the number of nodes that must be examined. Pre-reductions help doing this. For the same reason, it also helps if we quickly find a tree leaf that corresponds to an improved solution, because it can reduce (dynamically) the size of the tree. We found experimentally that searching the BB tree depth-first from the center is usually the most effective approach, because it permits one to find a shorter $\boldsymbol{v}_1$ faster. That is, at each level j, instead of scanning the interval for z_j from one size to the other, we scan it by starting with the z_j that is closest to the center, then we examine the second closest, etc. With this procedure, since the interval for z_t is symmetric about 0, we start with $z_t = 0$. For $z_t = 0$, the interval for z_{t-1} is also symmetric about 0 so we start with $z_{t-1} = 0$, and so on. Therefore the first visited branch corresponds to the trivial "solution" in which $z_t = \cdots = z_1 = 0$, and this solution is rejected. The second visited branch corresponds to $z_t = \cdots = z_2 = 0$ and $z_1 = 1$, i.e., $\boldsymbol{v} = \boldsymbol{v}_1$.

8.6 Inserting a new shorter vector in the basis

After we have found a new vector

$$\boldsymbol{v} = \sum_{i=1}^{t} z_i \boldsymbol{v}_i \tag{20}$$

shorter than all the current basis vectors $\boldsymbol{v}_i$, we may want to insert this new vector $\boldsymbol{v}$ in the basis to continue working with this modified basis and benefit from the new shorter vector. This is useful in particular when we reuse and extend this basis to a basis in $t+1$ dimensions as discussed in Section 5.4, in order to find a shortest vector in $t+1$ dimensions. In case $z_i = \pm 1$ for some i, we can simply replace $\boldsymbol{v}_i$ by $\boldsymbol{v}$ in the basis, since $\boldsymbol{v}_i$ can be expressed in terms of the other vectors as

$$\boldsymbol{v}_i = z_i \boldsymbol{v} - z_i \sum_{j=1,\, j\neq i}^{t} z_j \boldsymbol{v}_j.$$

This is the easy case.

If $|z_i| > 1$ for all the nonzero z_i's, a first option would be to take the set $\boldsymbol{v}, \boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ as a set of generating vectors and apply the modified LLL procedure mentioned in Section 5.3 to reduce this set into a basis that contains $\boldsymbol{v}$. This is basically the same as applying LLL.

A second option which is more direct and often faster operates by changing some basis vectors $\boldsymbol{v}_i$ in a way that when we express $\boldsymbol{v}$ in terms of the new $\boldsymbol{v}_i$'s, at least one of the coefficients z_i is ± 1. The idea of this approach comes from [1]. We first find the largest indices $i < j \leq t$ such that $|z_j| > 1$ and $|z_i| > 1$, and let $c_{i,j} = \gcd(z_i, z_j)$. There must be at least two nonzero z_i's and the gcd of all the nonzero z_i's must be 1, otherwise we could divide $\boldsymbol{v}$ by this gcd and obtain a shorter vector. When there are negative z_i's, we define $\gcd(z_i, z_j) = \gcd(|z_i|, |z_j|)$. We will apply transformations to $(z_i, \boldsymbol{v}_i)$ and $(z_j, \boldsymbol{v}_j)$ by applying the Euclidean algorithm until the new z_j is 0 and the new z_i is $\pm c_{i,j}$. The basis will be

changed along the way, but it will always remain a basis. Let $q = z_i/z_j$ (integer division, truncated toward 0). Note that $|z_i - qz_j| \leq \min(|z_j| - 1, |z_i|)$. We can write

$$z_i \boldsymbol{v}_i + z_j \boldsymbol{v}_j = (z_i - qz_j)\boldsymbol{v}_i + z_j(\boldsymbol{v}_j + q\boldsymbol{v}_i).$$

This shows that we can replace $\boldsymbol{v}_j$ by $\boldsymbol{v}_j + q\boldsymbol{v}_i$ in the basis and replace z_i by $z_i - qz_j$, and $\boldsymbol{v}$ is still given by (20). We then exchange $(z_i, \boldsymbol{v}_i)$ with $(z_j, \boldsymbol{v}_j)$. We repeat this process until $z_j = 0$. Then, the new z_i must be $\pm c_{i,j}$.

If $z_i = \pm 1$, we can now exchange $\boldsymbol{v}$ with this $\boldsymbol{v}_i$ in the basis and we are done. Otherwise, there must be some $i' < i$ for which $z_{i'} \neq 0$. We take the largest one, put $j = i$ and $i = i'$, and we repeat the same process as above. We must end up with $z_i = \pm 1$ at some point, because the gcd of the original nonzero z_j's must be 1. Then we can replace the corresponding (new) $\boldsymbol{v}_i$ by the new shorter vector $\boldsymbol{v}$ in the basis. Note that the lengths of the other basis vectors may increase during this process, and some of them may end up with very large coordinates. In our empirical experiments, however, this happened rarely. In most cases, we find $z_j = \pm 1$ right from the start. [3] The complete procedure is stated in Algorithm 4. It is implemented by the function `insertBasisVector` in `ReducerBB`.

Algorithm 4: : Inserting a new shorter vector in the basis

InsertVec$(\boldsymbol{V}, \boldsymbol{z})$:
  // To do ...
return;

8.7 Numerical illustration

We now provide a numerical example to illustrate what happens when we perform the BB procedure using the triangular vs Cholesky method for the bounds, for the primal and m-dual lattice, and with the L^1 and L^2 norms. The code used for this experiment is in `TestBBSmall`, in the examples. See also Section 12.3.

We consider the *primal lattice* associated with an LCG with modulus $m = 1021$ and multiplier $a = 73$ in $t = 4$ dimensions. The initial basis (2) for the rescaled lattice Λ_t is

$$\boldsymbol{V}^0 = \begin{pmatrix} 1 & 73 & 224 & 16 \\ 0 & 1021 & 0 & 0 \\ 0 & 0 & 1021 & 0 \\ 0 & 0 & 0 & 1021 \end{pmatrix}$$

[3] From Pierre: ** We need to make and report experiments with a variety of lattice types and dimensions to confirm that. The outcome may depend on the dimension and on the modulus m.

If we apply LLL with factor $\delta = 0.99$ to this $\boldsymbol{V}^{(0)}$, we obtain the reduced basis

$$\boldsymbol{V}^{\mathrm{R}} = \begin{pmatrix} 55 & -69 & 68 & -141 \\ 69 & -68 & 141 & 83 \\ -68 & 141 & 83 & -67 \\ -127 & -82 & 140 & 10 \end{pmatrix}$$

The first vector of this basis, $\boldsymbol{v} = (55, -69, 68, -141)$, is already a shortest vector in the lattice with respect to the L^2 norm. Its square length with respect to this norm is 32291 and its length is $\sqrt{32291} \approx 179.697$.

To verify that it is indeed a shortest vector for the L^2 norm, we apply the BB procedure, using either a Cholesky decomposition or a lower-triangular basis. For the Cholesky decomposition, we find that $\boldsymbol{V}^{\mathrm{R}}(\boldsymbol{V}^{\mathrm{R}})^{\mathrm{t}} = \boldsymbol{L}^{\mathrm{C}}(\boldsymbol{L}^{\mathrm{C}})^{\mathrm{t}}$ where

$$\boldsymbol{L}^{\mathrm{C}} = \begin{pmatrix} 179.697 & 0 & 0 & 0 \\ 0.050231 & 189.213 & 0 & 0 \\ 0.197331 & -0.236249 & 181.382 & 0 \\ 0.210059 & 0.214608 & 0.542857 & 172.58 \end{pmatrix}.$$

The upper bound for z_4 in (15) is then $z_4^{\max} = \lfloor b(2)/\ell_{t,t} \rfloor = \lfloor 179.697/172.58 \rfloor = 1$, so we only need to consider $z_4 = 0$ or 1. Starting with $z_4 = 0$ leads first to the trivial vector $\boldsymbol{z} = (0, 0, 0, 0)$, which must be discarded, and then to $\boldsymbol{z} = (1, 0, 0, 0)$, which corresponds to the current shortest vector. Starting with $z_4 = 1$ leads to no better candidate. There was only 5 calls to the recursive BB procedure in total, and no shorter vector was found.

With the lower-triangular basis approach, if we take the value closest to 0 when doing the modulo m operation, we find the basis

$$\boldsymbol{L}^{\mathrm{L}} = \begin{pmatrix} 1021 & 0 & 0 & 0 \\ 0 & 1021 & 0 & 0 \\ 0 & 0 & 1021 & 0 \\ -319 & 196 & 14 & 1 \end{pmatrix}$$

In the BB procedure, we use the same initial shortest vector square length for the L^2 norm, $b(2)^2 = 32291$. The bounds (15) on z_4 are $0 \le z_4 \le \lfloor b(2)/\ell_{t,t} \rfloor = 179$, so there are 180 values to consider. For most of them, no more than one or two values of z_3 must be examined. The total number of calls to the recursive BB procedure was 245. There was of course no shorter vector.

We now look for a shortest vector for the L^1 norm. The vector $(55, -69, 68, -141)$ is a shortest one with respect to the L^2 norm, and it has L^1 length 333. This means that $b(1) = 333$ is an upper bound on the length of a shortest vector for the L^1 norm, and we can use this $b(1)$ in the Cholesky decomposition approach for the L^1 norm. The Cholesky matrix $\boldsymbol{L}^{\mathrm{C}}$ is the same. The bounds on z_4 are again 0 and 1, so we only have two values to look at. Going down the tree, with $\boldsymbol{z} = (-1, 0, 1, 0)$ we find the vector $\boldsymbol{v} = (14, 1, 73, 224)$, whose L^1 length is 312. Then with $\boldsymbol{z} = (0, 0, -1, 1)$ we find $\boldsymbol{v} = (-196, -14, -1, -73)$, with

L^1 length of 284. This is a shortest vector for this norm. There was 16 calls to the recursive BB procedure.

With the triangular approach, we start with the same lower-triangular basis $\boldsymbol{L}^{\text{L}}$ as above, and use 333 as the length of a shortest known vector. The BB procedure finds the shorter vector $\boldsymbol{v} = (1, 73, 224, 16)$ with length 314 for $\boldsymbol{z} = (5, -3, 0, 16)$, and then $\boldsymbol{v} = (196, 14, 1, 73)$ with length 284 for $\boldsymbol{z} = (23, -14, -1, 73)$. The total number of calls to the recursive BB procedure was 386.

We can do the same experiment for the *m-dual lattice*. Its initial basis as in (3) is

$$\boldsymbol{W}^0 = \begin{pmatrix} 1021 & 0 & 0 & 0 \\ -73 & 1 & 0 & 0 \\ -224 & 0 & 1 & 0 \\ -16 & 0 & 0 & 1 \end{pmatrix}.$$

Applying LLL with factor $\delta = 0.99$ gives the reduced basis

$$\boldsymbol{W}^{\text{R}} = \begin{pmatrix} -2 & 2 & -1 & 5 \\ 0 & -5 & -3 & 1 \\ 1 & 2 & -5 & -3 \\ -5 & -3 & 1 & 0 \end{pmatrix},$$

whose first vector $\boldsymbol{w} = (-2, 2, -1, 5)$ has square length $b(2)^2 = 34$. The Cholesky matrix in this case is

$$\boldsymbol{L}^{\text{C}} = \begin{pmatrix} 5.83095 & 0 & 0 & 0 \\ -0.058823 & 5.90613 & 0 & 0 \\ 0.088235 & 0.349073 & 5.52131 & 0 \\ -0.235294 & 0.0438449 & -0.519209 & 5.3696 \end{pmatrix}.$$

Here, $z_4^{\max} = \lfloor \sqrt{34}/5.3696 \rfloor = 1$. By using it in the BB algorithm, we only need 6 calls to the recursive BB procedure to conclude that there is no shorter vector. With the lower-triangular basis approach, we find the basis

$$\boldsymbol{L}^{\text{L}} = \begin{pmatrix} 1021 & 0 & 0 & 0 \\ -73 & 1 & 0 & 0 \\ -5 & -3 & 1 & 0 \\ 0 & -5 & -3 & 1 \end{pmatrix}$$

We use again $b(2)^2 = 34$. The values of z_4 that must be examined here are only 0 to 5, but there is a total of 458 calls to the recursive BB procedure.

When looking for a shortest vector for the L^1 norm, we start with $\boldsymbol{w} = (0, -5, -3, 1)$, the second vector of the reduced basis, whose L^1 length is 9. When using the Cholesky decomposition with $b(1) = 9$, we find that this is a shortest vector after 16 calls to the recursive procedure. If we use the triangular basis approach instead, we need 500 calls.

We repeated the same experiment with the same parameters but in 8 dimensions, and then with $m = 1048573$ and $a = 29873$, in 4 dimensions. The results are summarized in

Table 1: Summary of the behavior of the BB algorithm for an LCG with modulus $m = 1021$ and $a = 73$ in 4 dimensions (above) and 8 dimensions (below), for the primal and m-dual lattices, with two different norms and the two decomposition methods. The last two columns give the length of the shortest vector and the number of calls to the recursive BB procedure when computing the shortest vector.

$m = 1021$, $a = 73$, 4 dimensions

primal/dual	norm	decomp	min length	num calls BB
primal	L^2	Cholesky	$\sqrt{32291}$	5
primal	L^2	triangular	$\sqrt{32291}$	245
primal	L^1	Cholesky	284	16
primal	L^1	triangular	284	386
dual	L^2	Cholesky	$\sqrt{34}$	6
dual	L^2	triangular	$\sqrt{34}$	458
dual	L^1	Cholesky	9	16
dual	L^1	triangular	9	500

$m = 1021$, $a = 73$, 8 dimensions

primal/dual	norm	decomp	min length	num calls BB
primal	L^2	Cholesky	$\sqrt{152466}$	8
primal	L^2	triangular	$\sqrt{152466}$	844
primal	L^1	Cholesky	948	1050
primal	L^1	triangular	948	2748
dual	L^2	Cholesky	$\sqrt{6}$	18
dual	L^2	triangular	$\sqrt{6}$	1576
dual	L^1	Cholesky	4	216
dual	L^1	triangular	4	708

$m = 1048573$, $a = 29873$, 4 dimensions

primal/dual	norm	decomp	min length	num calls BB
primal	L^2	Cholesky	10036.8	4
primal	L^2	triangular	10036.8	10195
primal	L^1	Cholesky	18910	4
primal	L^1	triangular	18910	19260
dual	L^2	Cholesky	14.8	4
dual	L^2	triangular	14.8	7116
dual	L^1	Cholesky	21	4
dual	L^1	triangular	21	6204

Table 1. We see that in all cases, the triangular approach leads to more calls of the recursive BB procedure than the Cholesky decomposition. The L^1 norm also tends to require more recursive calls than the L^2 norm. More extensive experiments in a wider range of dimensions, and timing comparisons, are reported in Section 12.8.

Aside from the larger number of recursive calls, another insidious problem often occurs with the triangular approach, especially when m is large: the center c_j of the interval in (13) can easily get very far away from 0, which can cause numerical instabilities, as shown in the next example.

Example 3. For $m = 1048573$ and $a = 29873$, for the dual lattice in 4 dimensions, we have the lower-triangular basis

$$\boldsymbol{L}^{\text{L}} = \begin{pmatrix} 1048573 & 0 & 0 & 0 \\ -29873 & 1 & 0 & 0 \\ -166086 & 203765 & 1 & 0 \\ 400114 & 331129 & -289737 & 1 \end{pmatrix}.$$

With the L^2 norm, the shortest vector obtained after LLL had square length 219, so the upper bound on z_4 is $\lfloor\sqrt{219}\rfloor = \lfloor 14.7986\rfloor = 14$. When we put $z_4 = 14$, the third coordinate of our candidate vector $\boldsymbol{w}$ will be $z_3 - 14 \times 289737 = z_3 - 4056318$ and the sum of squares of its last two coordinates will be $14^2 + (z_3 - 4056318)^2$, which must not exceed 219. For this, we must have $4056314 \le z_3 \le 4056322$. Then when we take $z_3 = 4056320$ for instance, with a similar calculation we find that we must have $-826540680610 \le z_2 \le -826540680602$. In large dimensions, the numbers sometimes keep increasing like this and this leads to numerical issues. This occurs mostly when the lower-triangular basis vectors have large entries. It gets worse when m is larger and gets a bit worse if the entries below the diagonal are reduced modulo m in a way that they are non-negative instead of taking the value closest to 0 as done by default in Lattice Tester. See Section 12.7 for more examples of what can happen with this.

9 Normalized Measures of Uniformity

The length of the shortest nonzero vector in a lattice or its dual gives information about the uniformity of the points, but what is a "good" value for this length? It depends on the lattice density and also on the dimension. To obtain a "normalized" measure that is easier to interpret and can be compared across different densities and numbers of dimensions, it is common to take the ratio between the actual shortest vector length and the largest possible value that can be obtained for the given density and dimension (or an estimate of that value). This gives a number between 0 and 1, and we want it to be close to 1 as much as possible, because we want the shortest vector to be as large as possible. This measure can also be computed for several projections of the lattice on subsets of coordinates, and then we can take the worst-case or some (weighted) average over the selected projections as a FOM for this lattice. We now explain what support Lattice Tester offers for this. We first discuss

upper bounds on the shortest vector length, to define the normalized measures. Such bounds were already mentioned in Section 7.3. In Section 10, we look at figures of merit that take several projections into account.

Bounds for the Euclidean norm. In this section, until stated otherwise, we assume that the vector lengths are measured with the Euclidean norm, and we consider the (primal) lattice L_t with density η_t in t dimensions. We can view a lattice as a way of packing the space by non-overlapping spheres of radius $d_t/2$, with one sphere centered at each lattice point. We have η_t spheres per unit of volume. If we rescale by the factor $2/d_t$ so that the radius of each sphere is 1, we obtain $\delta_t = (d_t/2)^t \eta_t$ unit spheres per unit volume. This number δ_t is called the *center density* of the lattice. The largest possible center density of unit spheres for a general t-dimensional lattice is $\delta_t^* = (\gamma_t/4)^{t/2}$, where γ_t is the *Hermite constant* for dimension t [8, 17, 69]. This gives the following upper bound $d_t^*(\eta_t)$ on d_t for a general lattice L_t of density η_t:

$$d_t \le d_t^*(\eta_t) \stackrel{\text{def}}{=} 2(\delta_t^*/\eta_t)^{1/t} = \gamma_t^{1/2} \eta_t^{-1/t}. \tag{21}$$

For $t \ge k$, under the assumption that L_t has density $\eta_t = m^k$, the right side becomes $\gamma_t^{1/2} m^{-k/t}$, whereas for the dual lattice L_t^* it becomes $\gamma_t^{1/2} m^{k/t}$. This is for the *non-rescaled* lattice L_t. The Hermite constants γ_t are known (proved) exactly only for $t \le 8$ and $t = 24$. These known values are given in Table 2.

Table 2: The Hermite constants for $t \le 8$ and $t = 24$ dimensions

t	2	3	4	5	6	7	8	24
γ_t	$(4/3)^{1/2}$	$2^{1/3}$	$2^{1/2}$	$8^{1/5}$	$(64/3)^{1/6}$	$64^{1/7}$	2	4

In view of (21), a good way to normalize d_t to a value between 0 and 1 is by taking $d_t/d_t^*(\eta_t)$. This is convenient for comparing uniformity measures for different values of t and η_t, and we will use that to define figures of merit that take several projections into account. For $t > 8$ and $t \ne 24$, where the exact value of γ_t is unknown, we can replace it by an approximation or by a bound on γ_t. Mächler and Naccache [66] suggest a reasonable formula for the exact values of γ_t for $1 \le t \le 24$. The formula matches the values known so far, but it remains a conjecture. It could nevertheless be used to compute reasonable normalization constants. Another option is to use proved lower or upper bounds on the Hermite constants γ_t. We now discuss various available bounds of this type.

Recall that $\delta_t^* = (\gamma_t/4)^{t/2}$, so $\gamma_t = 4(\delta_t^*)^{2/t}$. Conway and Sloane [8], in Table I.1 of their preface, give the highest center densities δ_t currently attained by known lattice constructions in up to 48 dimensions and in a few larger dimensions. These δ_t are lower bounds on δ_t^*, so $\gamma_t^{\mathrm{B}} = 4\delta_t^{2/t}$ is a *lower bound on* γ_t. This Table I.1, as well as Table 1.2 on page 15 of [8], also give the largest center densities reached by any known construction, but some of these constructions are not lattices. In 10, 11, ans 13 dimensions, for example, the densest known

packings are not lattices. Note that to bound γ_t, we can only consider the center densities of lattice constructions.

One important type of high-density lattice construction is the *laminated lattice*, discussed in Chapter 6 of [8]. The densest known lattice packings in dimensions 1 to 26, except for dimensions 10 to 13, attained by these constructions. By taking the center density δ_t of the best laminated lattice in t dimensions, we get the lower bound $\gamma_t \geq \gamma_t^{\mathrm{L}} = 4\lambda_t^{-1/t}$, where the constants $\lambda_t = \delta_t^{-2}$ are given in [8, Table 6.1, page 158] for $t \leq 48$, For $t \leq 8$, one has $\gamma_t^{\mathrm{L}} = \gamma_t$.

The Minkowski-Hlawka theorem [21, 80] says that there exists lattices with density satisfying $\delta_t \geq \zeta(t)/(2^{t-1}V_t)$ where $\zeta(t) = \sum_{k=1}^{\infty} k^{-t}$ is the Riemann zeta function and $V_t = \pi^{t/2}/(t/2)!$ is the volume of a t-dimensional sphere of radius 1. This provides a lower bound on γ_t given by

$$\gamma_t^{\mathrm{Z}} = 4[\zeta(t)/(2^{t-1}V_t)]^{2/t}.$$

Note that if we replace γ_t in (21) by one of the lower bounds just given, we no longer have an upper bound on d_t, but only a lower bound on the upper bound on d_t, and then it is conceivable that the normalized value could exceed 1. If this occurs with the bound γ_t^{B}, than it would mean that we have found a denser lattice for this t than the best ones known so far. The author believes that this is unlikely to happen, and that if it happens the change will be small, so using γ_t^{B} to approximate γ_t seems to be the best choice.

If we insist on having a true *upper bound on* γ_t, an old upper bound valid for all t is [4]:

$$\gamma_t \leq (2/\pi)(\Gamma(2 + t/2))^{2/t}.$$

Another upper bound is obtained via the bound of Rogers on the density of sphere packings in general (lattices or not) [8]. By using this bound for lattice packings, we get

$$\gamma_t^{\mathrm{R}} = 4 \times 2^{2R(t)/t}$$

where $R(t)$ can be found in Table 1.2 of [8] for $t \leq 24$, and can be approximated with $O(1/t)$ error and approximately 4 decimal digits of precision, for $t \geq 25$, by

$$R(t) = \frac{t}{2}\log_2\left(\frac{t}{4\pi e}\right) + \frac{3}{2}\log_2(t) - \log_2\left(\frac{e}{\sqrt{\pi}}\right) + \frac{5.25}{t + 2.5}.$$

One also has $\gamma_t^{\mathrm{R}} = (V_t\theta_t)^{2/t}$ where θ_t is given by the last equation of Rogers [74], and V_t is the volume of a unit sphere in t dimensions. Table 1 in [34] gives the ratio $(\gamma_t^{\mathrm{L}}/\gamma_t^{\mathrm{R}})^{1/2}$, of the lower bound on d_t based on γ_t^{L} over the upper bound based on γ_t^{R}, for $1 \leq t \leq 48$. This ratio tends to decrease with t, but not monotonously.

Tighter upper bounds on sphere packing densities (in general) were obtained via linear programming inequalities by Cohn and Elkies [7], for up to 36 dimensions. These bounds are also given in Table 2 of [6]. They provide upper bounds on γ_t which we denote γ_t^{C}. Our best available upper bounds on the unknown γ_t's (for $t > 8$ and $t \neq 24$) are then γ_t^{C} for $t \leq 36$ and γ_t^{R} for $t > 36$.

Simple upper bounds that are linear in t and are valid for all $t \geq 2$ have been proposed in [85, 84] and references given there: $\gamma_t \leq 2t/3$, $\gamma_t \leq 1 + t/4$, $\gamma_t \leq (t+6)/7$, $\gamma_t \leq t/8 + 6/5$, and $\gamma_t \leq t/8.5 + 2$. All these bounds increase linearly in t, and they are given by decreasing order of slope. The latest one is the sharpest when $t \geq 109$.

We denote by ℓ_t the Euclidean length of the shortest nonzero vector in the *dual lattice* L_t^*. Recall that $1/\ell_t$ corresponds to the distance between successive parallel hyperplanes that contain all the lattice points in L_t [11, 25]. For a good quality lattice, we want this distance to be small, so we want ℓ_t to be large [25, 47]. Since the density of L_t^* is $\eta_t^* = 1/\eta_t$, we have the upper bound

$$\ell_t \leq \ell_t^*(\eta_t) := \gamma_t^{1/2} \eta_t^{1/t}. \tag{22}$$

We can then normalize ℓ_t to $\ell_t / \ell_t^*(\eta_t) \in (0, 1]$. Again, γ_t can be replaced by an approximation, good scores are close to 1, and bad ones are close to 0.

Bounds for the L^p norm for all $p > 0$. In general, for $0 < q < p$, we always have

$$\|\boldsymbol{v}\|_p \leq \|\boldsymbol{v}\|_q \leq t^{(p-q)/pq} \|\boldsymbol{v}\|_p, \tag{23}$$

which follows from Hölder's inequality. By taking $q = 2$, the first inequality says that for $p \geq 2$, $\|\boldsymbol{v}\|_p \leq \|\boldsymbol{v}\|_2$, which implies that the bounds (21) and (22) are also valid for the L^p norm for any $p \geq 2$. By taking $p = 2$ and $0 < q < 2$, the second inequality tells us that

$$\|\boldsymbol{v}\|_q \leq t^{(2-q)/2q} \|\boldsymbol{v}\|_2. \tag{24}$$

So the bounds (21) and (22) can be transformed into bounds valid for the L^q norm for $0 < q < 2$ by multiplying their right-hand side by $t^{(2-q)/2q}$, or equivalently replacing γ_t by $\gamma_t^{(p)} = t^{(2-q)/q} \gamma_t$ for each t. These transformed bounds remain upper bounds when they contain the exact values of γ_t or upper bounds on γ_t. But when γ_t is approximated by a lower bound such as γ_t^{B} or γ_t^{L}, then the transformed bounds are neither upper bounds nor lower bounds, they are only approximations.

For the special case of the L^1 norm ($q = 1$), this gives $\gamma_t^{(1)} = t\gamma_t$ and then

$$d_t^{(1)} \leq (t\gamma_t)^{1/2} \eta_t^{-1/t} \quad \text{and} \quad \ell_t^{(1)} \leq (t\gamma_t)^{1/2} \eta_t^{1/t}, \tag{25}$$

where $d_t^{(1)}$ and $\ell_t^{(1)}$ are the shortest vector lengths with the L^1 norm in the primal and m-dual lattices. Recall that $\ell_t^{(1)} - 1$ corresponds to the minimal number of hyperplanes that contain $L_t^* \cap (0,1)^t$, the dual lattice points that are in the open unit hypercube, as we saw in Section 3. If we replace γ_t in (25) by an upper bound, the inequalities still hold, but if we replace γ_t by γ_t^{B} for example, they become only approximate (not proven inequalities), although the right side can still be a good choice the normalization.

Alternative upper bounds on $d_t^{(1)}$ and $\ell_t^{(1)}$ are obtained by applying the general convex body theorem of Minkowski [13, 67]. It says that for a lattice of density η_t, the shortest vector length cannot exceed $(t!/\eta_t)^{1/t}$. This implies that by taking $\gamma_t^{\mathrm{M}} = (t!)^{2/t}$,

$$d_t^{(1)} \leq d_t^{*(1)}(\eta_t) \stackrel{\text{def}}{=} (\gamma_t^{\mathrm{M}})^{1/2} \eta_t^{-1/t} \quad \text{and} \quad \ell_t^{(1)} \leq \ell_t^{*(1)}(\eta_t) \stackrel{\text{def}}{=} (\gamma_t^{\mathrm{M}})^{1/2} \eta_t^{1/t}. \tag{26}$$

The upper bounds (25) and (26) can then be used to normalize the shortest vector lengths for the L^1 norm, with γ_t in (25) replaced by an approximation when $t > 8$, just like for the bounds in (25).

The normalizer classes. In Lattice Tester, the class `Normalizer` and its several subclasses whose names start with `Norma...` compute normalization factors that correspond to different approximations of the constants γ_t, where η_t is the density of the lattice L_t (non rescaled) in t dimensions. These normalization factors are then modified to obtain bounds for the rescaled lattice Λ_t. To make sure that we can handle lattice densities that are very large or very small, the computations are done using the log (in natural basis) of the normalization constants, then we apply the exponential function.

The bounds are computed by assuming that the lattice L_t has *density $\eta_t = m^k$ for $t > k$ and m^t for $t \le k$*. This corresponds to the lattice L_t generated by an MRG with modulus m and order k. We *assume* that this lattice is rescaled by a factor m, so the *rescaled lattice* Λ_t in t dimensions has density

$$\tilde{\eta}_t = \eta_t / m^t = \begin{cases} 1 & \text{for } t \le k; \\ m^{k-t} & \text{for } t > k. \end{cases}$$

See our discussion at the end of Section 3.2. When $t \gg k$ and m is large, this density is very small, which means that the shortest nonzero vector is usually very large. We denote by $\tilde{d}_t$ the L^2 length of a shortest nonzero vector in Λ_t. In the software, we compute the bounds for the rescaled lattice because this is the one we use when working with the primal lattice. When we work with the dual, the rescale makes no difference, because the m-dual is not rescaled. The m-dual of Λ_t, which is also the dual of L_t, has density

$$\eta_t^* = \begin{cases} m^{-t} & \text{for } t < k; \\ m^{-k} & \text{for } t \ge k. \end{cases}$$

For the projections in $t \le k$ dimensions, tighter bounds can be obtained by observing that all vectors in Λ_t and L_t^* must have integer coordinates. In the best case, $\Lambda_t = \mathbb{Z}^t$, the set of all integer vectors, and then the shortest nonzero vector must have length 1, while the bound $d_t^*(\tilde{\eta}_t) = d_t^*(1)$ is $\gamma_t^{1/2} > 1$. The smaller bound comes from the fact that the points are forced to have a rectangular grid structure in the best case, and this is not optimal. In this case, we may prefer to use 1 instead of $\gamma_t^{1/2}$ for the normalization. The $\gamma_t^{1/2}$ factor measures the gain obtained by allowing general non-integral lattices instead of just rectangular grids. For the m-dual, regardless of the dimension, the vectors $m\boldsymbol{e}_i$ always belong to the lattice, which means that the length of a shortest nonzero vector can never exceed m. Therefore, the upper bound $\ell_t^*(\eta_t) = \gamma_t^{1/2} m$ for $t \le k$ can be replaced by m. This is what we compute in `Normalizer.h`.

All the upper bounds discussed so far also hold for the length $\tilde{d}_I$ of a shortest vector in the projection Λ_I of Λ_t over a set I of s coordinates, and for the length ℓ_I of a shortest vector in its m-dual L_I^*, with t replaced by s, if we assume that the density of the lattice Λ_I is the

same as that of Λ_s, namely m^{k-s} for $s > k$ and 1 for $s \le k$. For a subset I of cardinality s, with the L^2 norm, an upper bound on $\tilde{d}_I$ is then

$$\tilde{d}_s^*(m,k) = d_s^*(\tilde{\eta}_s) = \begin{cases} 1 & \text{for } s \le k; \\ \gamma_s^{1/2} m^{1-k/s} & \text{for } s > k. \end{cases} \tag{27}$$

The log of this bound is

$$\ln \tilde{d}_s^*(m,k) = \begin{cases} 0 & \text{for } s \le k; \\ 0.5 \ln \gamma_s + (1-k/s) \ln m & \text{for } s > k. \end{cases} \tag{28}$$

Likewise, an upper bound on ℓ_I for the m-dual lattice L_I^* is

$$\tilde{\ell}_s^*(m,k) = \begin{cases} m & \text{for } s \le k; \\ \ell_s^*(\eta_s) = \gamma_s^{1/2} m^{k/s} & \text{for } s > k. \end{cases} \tag{29}$$

The log of this bound is

$$\ln \tilde{\ell}_s^*(m,k) = \begin{cases} \ln m & \text{for } s \le k; \\ 0.5 \ln \gamma_s + (k/s) \ln m & \text{for } s > k. \end{cases} \tag{30}$$

The subclasses of `Normalizer` compute approximations of these bounds (for the L^2 norm) by using approximations of the γ_s's. They also compute approximate bounds for the L^1 norm, simply by replacing the approximations of γ_s by approximations of $\gamma_s^{(1)}$. See Section 11.8.

Losing points in projections. It happens sometimes that the density of a projection Λ_I over a set I of s coordinates is smaller than 1 for $s \le k$ or smaller than m^{k-s} for $s > k$. We give a simple numerical example of that below. This means that points are projected onto each other and Λ_I is a bad projection because it has a smaller density than the original lattice. Note that this happens if and only if the linear transformation from the initial state $(x_0, \ldots, x_{k-1})$ to the output vector $(u_{i_1}, \ldots, \ldots u_{i_s})$ is *not* one-to-one, i.e., when the corresponding matrix does not have full rank. Note that the true density of the projection can be computed easily by calculating the determinant of a triangular basis, but we do not really want to normalize by using the true density of the projection in this case, because a smaller density already means that the quality of the lattice is bad.

When a projection has smaller density, the density of the *m-dual* will be larger, so the shortest vector in the dual space will typically be smaller, because the upper bound on its length is actually smaller than the value of $\ell_I^*(\eta_t)$ that we compute. Then the standardized measure $\ell_I/\ell_I^*(\eta_t)$ that we compute will be smaller and the quality of the lattice will be deemed very poor, as it should.

If we use only the lengths of the shortest vectors in the *primal* space, we have the opposite. If the density has been reduced, then the standardized measure $\tilde{d}_I/d_I^*(\tilde{\eta}_t)$ will be too large, eventually larger than 1, so the bad projection is likely to be left undetected. Therefore, when looking for bad projections Λ_I using the standardized measures, we better stick to the

dual space. When we observe a standardized measure larger than 1 for a projection in the primal lattice, it is most likely because the density has been reduced by doing the projection, and not because of an error in the software or in the bounds.

For a rank-1 lattice whose coefficients a_j are all nonzero and relatively prime to m, all projections have the same density m, so this lower-density problem cannot occur. This holds for example for a LCG whose multiplier a is relatively prime to m.

Example 4. Normalized values that exceed 1. This is a simple example in which a projection has a smaller density than the full lattice. Consider an MRG with order $k = 3$, modulus $m = 13$, and multipliers $(a_1, a_2, a_3) = (7, 0, 4)$. See the LatMRG guide [40] for more details about MRGs. For $t = 4$ dimensions, the standard basis matrix $\boldsymbol{V}$ in Eq. (13) of the LatMRG guide is

$$\boldsymbol{V} = \begin{pmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 7 \\ 0 & 0 & 0 & 13 \end{pmatrix}$$

This lattice has density $1/\det(\boldsymbol{V}) = 1/13$, while its projection over the first one, two, or three coordinates has density 1. Many of the two- or three-dimensional projections may also have density 1, and the standardized measures are computed by assuming that they have that density. When a t-dimensional integral lattice has density 1, it must be $\mathbb{Z}^t$, and the length of a shortest nonzero vector must be 1.

By taking columns 1, 3 and 4 of this matrix $\boldsymbol{V}$ to obtain a set of generating vectors, and applying the triangulation method, we obtain the following triangular basis for the projection Λ_I over coordinates $I = \{1, 3, 4\}$:

$$\boldsymbol{V}_I = \begin{pmatrix} 1 & 0 & 4 \\ 0 & 1 & 7 \\ 0 & 0 & 13 \end{pmatrix}$$

A shortest vector in this projection is $(0, 2, 1)$, whose square length is 5. It is minus the last vector plus twice the second one. But the density of this projection is $1/13$, not 1. That is, the lattice L_I for this projection has only 13^2 points in the unit cube $[0, 1)^3$, not 13^3 points as we may have expected. We have 13 times fewer points, which is not good. If we compute $\tilde{d}_I/\tilde{d}_I^*(\tilde{\eta}_t)$ by assuming that the density is 1, we obtain approximately 2.236, which is larger than 1.

10 Figures of Merit That Examine Projections

For any normalized measure of uniformity for L_t or Λ_t as defined in the previous section, one can compute this measure not only for L_t but also for L_I for any subset of indices $I = \{i_1, \ldots, i_s\} \subseteq \{1, \ldots, t\}$. This can be done for a selected class $\mathcal{I}$ of subsets I. For each $I \in \mathcal{I}$, the uniformity measure is normalized to a value between 0 and 1. Then one can give

a weight ω_I to each subset $I \in \mathcal{I}$ and take the (possibly weighted) sum or the worst case (minimum) of the normalized values as a FOM for the lattice L_t. The same can be done for the m-dual lattice L_t^* and projections L_I^*. This can be done for either the L^2 or L^1 norm.

As a concrete example of a standard FOM available in Lattice Tester, for each selected I with $|I| = s$, we can take the Euclidean length ℓ_I of the shortest nonzero vector in the dual lattice L_I and normalize it to $\ell_I/\tilde{\ell}_s^*(m,k)$ with $\tilde{\ell}_s^*(m,k)$ defined in (29). Here we take the density that L_I has when there is no loss of points in the projections, as explained at the end of the previous section. That is, we assume a density of $\tilde{\eta}_I = m^{\min(1,s-k)}$ for Λ_I and $\eta_I^* = 1/\eta_I = m^{-\min(s,k)}$ for the m-dual L_I^*. The unknown γ_t's are replaced by estimates such as γ_t^{L} or γ_t^{B}, for example. For an integer $d \geq 1$ and a vector of arbitrary non-negative integers $\boldsymbol{t} = (t_1, \cdots, t_d)$, we define the general weighted worst-case FOM

$$M_{\boldsymbol{t}} = M_{t_1,\ldots,t_d} = \min_{1 \leq s \leq d} \min_{I \in S_s(t_s)} \frac{\ell_I}{\omega_I \, \tilde{\ell}_s^*(m,k)} \tag{31}$$

where $s = |I|$,

$$S_1(t_1) = \{I = \{1,\ldots,s\} \mid d+1 \leq s \leq t_1\}, \tag{32}$$

$$S_s(t_s) = \{I = \{i_1,\ldots,i_s\} \mid 1 \leq i_1 < \cdots < i_s \leq t_s\}, \tag{33}$$

and $\omega_I > 0$ for all I considered. Note that we divide by the weight, so the term in (31) that corresponds to a set I having a small weight is inflated, which reduces its importance because the minimum then becomes less likely to be reached by this term. Giving a large weight, in contrast, increases the importance. This FOM takes the worst case over all the projections over s successive dimensions for $d < s \leq t_1$, and over sets of possibly non-successive coordinates that are not too far apart in up to d dimensions. Note that if $t_s \geq s$, then $S_s(t_s)$ contains $\{1,\ldots,s\}$, and if this holds for all $2 \leq s \leq d$, then there is no loss in starting $S_1(t_1)$ with the projection $I = \{1,\ldots,d+1\}$, because the projections over $s \leq d$ coordinates are already accounted for. If we want to consider only the projections in $S_1(t_1)$ for a given $d > 1$, it is allowed to select that d and take $t_2 = \cdots = t_d = 0$.

This type of FOM was proposed in [35, 37, 50], for example, and is also available in LatNet Builder and LatMRG [46, 49]. It is usually computed for the m-dual lattice projections Λ_I^* [47]. The special case of (31) with $d = 1$ and unit weights,

$$M_{t_1} = \min_{2 \leq s \leq t_1} \ell_s/\ell_s^*, \tag{34}$$

has been used for ranking and selecting LCGs and MRGs [15, 31, 33, 34, 44, 56, 59, 60].

This FOM can also be defined in the primal lattice, and it works for either the L^2 or L^1 norm. For the primal lattice, it suffices to replace $\ell_I/\tilde{\ell}_s^*(m,k)$ by $\tilde{d}_I/\tilde{d}_s^*(m,k)$ for each I in (31). For the L^1 norm, the bounds in the denominators are modified appropriately as explained in Section 9. These FOMs are implemented in Lattice Tester in `FigureOfMeritM.h` and `FigureOfMeritMDual.h`.

To get an idea of how many terms there are in (31), note that the set $S_1(t_1)$ has cardinality $t_1 - d$, $S_2(t_2)$ has cardinality $\binom{t_2}{2} = t_2(t_2-1)/2$, and more generally $S_s(t_s)$ in (33) has

cardinality $\binom{t_s}{s}$, for $2 \le s \le d$. Therefore $M_{t_1,\ldots,t_d}$ is a worst case over $(t_1 - d) + \sum_{s=2}^{d} \binom{t_s}{s}$ projections. This number increases quickly with d unless the t_s are very small. For example, if $d = 4$ and $t_s = 32$ for $s = 1, \ldots, 4$, there are $28 + 496 + 4,960 + 35,960 = 41,444$ projections. The last three terms in this sum are the projections of order 2, 3, and 4. Note the very large number of projections of order 4. Also, the number of projections over successive coordinates (only 28) is very small compared to the total number. When too many projections are considered, we inevitably find some bad ones, so the worst-case figure of merit is (practically) always small, and can no longer distinguish between good and mediocre behavior in the most important projections. Moreover, the time to compute $M_{t_1,\ldots,t_d}$ increases with the number of projections. We should therefore make a compromise and consider only the projections deemed important. We suggest using the criterion (31) with d equal to 4 or 5, and t_s decreasing with s.

When projections of certain orders are too numerous, we may want to give smaller weights to these projections so they do not occupy the whole stage. We could think of giving weights ω_I that decrease with s, the cardinality of I, and perhaps also with the spacing $t_s - t_1$. Various ways of specifying the weights ω_I for subsets of coordinates are implemented in Lattice Tester in subclasses of the abstract class `Weights`. These weights are routinely used when constructing rank-1 lattice rules for quasi-Monte Carlo [53, 51, 49], usually to give more weight to the low-order projections in FOMs defined as a sum over the projections, but they are typically not used for RNGs.

The most general type of weights are *projection-dependent weights*, which permit one to specify a separate weight ω_I for each projection I. However, this quickly becomes impractical when the number of dimensions increases, since there are $2^t - 1$ values to specify in t dimensions. At the other extreme, *uniform weights* give the same weight $\omega_I = 1$ to all the projections I. They are the most restrictive ones. In between, we find *order-dependent weights*, for which ω_I depends only on the cardinality of I: we select real numbers Γ_i for $i = 1, \ldots, t$ and put $\omega_I = \gamma_{|I|}$ for each I. For *product weights*, we select a real number γ_i for each coordinate i, usually $\gamma_i \le 1$, and we put $\omega_I = \prod_{i \in I} \gamma_i$ for each I. For each of the last two types, only t real numbers have to be selected. Their multiplicative combination gives the *product-and-order-dependent (POD) weights*, for which $\omega_I = \Gamma_{|I|} \prod_{j \in I} \gamma_j$. All these types of weights are implemented in classes having the corresponding names.

When the lattice comes from a linear RNG based on a recurrence (like for an LCG, an MRG, a Korobov lattice rule, etc.), we say that we have a *recurrence-based point set* [54, 36]. In this case, shifting all coordinate indices of a set I by the same constant does not change the point set: we say that the point set is *projection-stationary*. Then we can impose the extra condition $i_1 = 1$ in the definition of $S_s(t_s)$ without changing the value of $M_{t_1,\ldots,t_d}$ in (31). That is, for $s \ge 2$, we can replace $S_s(t_s)$ by the smaller set

$$S_s^{(1)}(t_s) \;=\; \{I = \{i_1, \ldots, i_s\} \mid 1 = i_1 < \cdots < i_s \le t_s\}. \tag{35}$$

In our implementations, the user has the choice of imposing this condition or not. We write the FOM $M_{\boldsymbol{t}}^{(1)} = M_{t_1,\ldots,t_d}^{(1)}$ when the condition is imposed.

The set $S_s^{(1)}(t_s)$ in (35) has cardinality $\binom{t_s-1}{s-1}$. In this case, $M_{t_1,\ldots,t_d}$ is a worst case over $(t_1-d)+\sum_{s=2}^{d}\binom{t_s-1}{s-1}$ projections, which can be much smaller than for the earlier example. For $d=4$ and $t_s=32$ for all s, we now have $28+31+465+4495=5019$ projections, compared to $41,444$ in the earlier case. The total number is smaller, but the projections of order 4 still dominate, so one should probably take a smaller value for t_4.

As another example, if $d=5$ and $(t_1,\ldots,t_d)=(32,32,16,12,10)$, then there are $27+31+120+220+210=608$ projections. Here, the number of projections of different orders is more balanced.

A different type of FOM offered in Lattice Tester is the following one. Let q_I be the maximum of the Beyer quotients of all Minkowski-reduced lattice bases of L_I, and denote $q_{\{1,\ldots,t\}}$ by q_t. We prefer values of q_I close to 1. Similar to (31) and (34), we can define

$$Q_{\boldsymbol{t}} = Q_{t_1,\ldots,t_d} = \min_{1\le s\le d}\ \min_{I\in S_s(t_s)} q_I \tag{36}$$

and

$$Q_{t_1} = \min_{2\le t\le t_1} q_t. \tag{37}$$

Although the q_t's in this FOM can be computed with Lattice Tester, computing q_t is much more time consuming than computing the spectral test and we do not think that it is more relevant for measuring the uniformity. We have a legacy implementation of (37) mainly because one referee requested it for the paper [44]. This measure is used for the primal lattice [47].

The classes `FiguresOfMeritM` and `FiguresOfMeritMDual` in Lattice Tester implement the FOM (31) for the primal and the m-dual, respectively. One important practical consideration is that these FOM functions must take as inputs some thresholds outside of which the evaluation procedure can be stopped prematurely. For instance, if we are making a search for MRG parameters that maximize the FOM, we will give a minimal value `lowBound` such that as soon as we encounter a projection for which the FOM is below this value, we stop the FOM calculation and exit immediately. This is important, because when searching for RNGs with a good lattice structure, for example, we typically want to examine millions of candidates, and we can reject the great majority of candidates (say $> 99.9\%$) after looking at the spectral test results (or even just LLL reductions) in relatively low dimensions, so there is no need to do the expensive high-dimensional evaluations. This can make a huge difference in CPU times (see the example in Section 12.11).

Because of this frequent early exit, the order in which the two terms in (31) are evaluated can have a significant impact on the performance. Since the evaluation cost increases rapidly with the dimension, we prefer to evaluate the low-dimensional projections first, in the hope that the high-dimensional ones will have to be evaluated only in rare cases. Specifically, we start with the projections I in $s=2$ dimensions, then $s=3$, etc., and we finish with the projections over successive coordinates, in the set $S_1(t_1)$. Numerical experiments have confirmed that this can make a important difference.

In (31), one could also replace the min by a sum and reverse the terms in the fraction. We would then want to minimize that FOM instead of maximizing it. One drawback would be that most of the terms would have to be computed before we know that the FOM is too large, so the gain from early exit is likely to be smaller.

11 Main facilities provided by Lattice Tester

11.1 General overview

Here we give a tour of what the software provides and how to use it. The goal is to tell the user where to find the most important ingredients. Detailed examples are provided in Section 12.

We start with a general overview of all the files, then we discuss the important ones in slightly more details. The main type of object handled by this software is an *integral lattice*, represented in the base class `IntLattice`. Each `IntLattice` object has a dimension t, a scaling factor m, a basis, an m-dual basis (optional), a norm to measure vector lengths, etc. The class `IntLattice` contains methods to manipulate the lattice and to perform certain operations such as to compute and store the norms of the basis and m-dual basis vectors, permute the basis vectors, sort them by length, etc.

The abstract class `IntLatticeExt` extends `IntLattice` and contains (additional) virtual methods that must be defined in its subclasses because they depend on how the lattices are constructed. It is a skeleton for the specialized subclasses that define specific types of lattices. There are virtual methods to construct a basis or an m-dual basis, to extend the current basis (or its m-dual) by one coordinate, to construct the lattice defined as the projection of the full lattice on a subset of coordinates indices, and recompute a basis for different numbers of dimensions and subsets of coordinates. One subclass of `IntLatticeExt` offered in Lattice Tester is `Rank1Lattice`, whose objects are the lattices of rank 1 commonly used for lattice rules in quasi-Monte Carlo integration. Other subclasses are defined in LatMRG.

The file `BasisConstruction` provides static methods to construct a lattice basis from an arbitrary set of generating vectors, to compute the m-dual of a given basis, and to compute a basis for the projection of a lattice over a given subset of coordinates. The constructed basis can be LLL-reduced, or can be upper- or lower-triangular. In many cases, a lattice basis can be constructed directly by exploiting the definition and structure of the lattice. This is done in the subclasses of `IntLatticeExt`.

The file `ReducerStatic` provides static functions to reduce a lattice basis via LLL or BKZ by using modified versions of NTL functions, using the L^2 norm to measure vector lengths. For the LLL and BKZ algorithms, we use slightly modified versions of the NTL implementations, whose header files are `LLL_FPInt.h` and `LLL_lt.h`. Our versions permit one to recover the square length of the basis vectors and always return the shortest vector in first place. These algorithms are used by `BasisConstruction` and `ReducerStatic`.

The class `ReducerBB` offers tools to compute a shortest lattice vector via the BB algorithm, with either the L^2 norm or the L^1 norm. It also offers a procedure to compute a Minkowski-reduced basis and the Beyer quotient. All these tools require that a lattice basis has already been constructed.

The length of the shortest vector can be normalized (usually to a value between 0 and 1) by using one of the normalizations implemented in the `Norma`$\cdots$ subclasses of `Normalizer`. There are several possibilities, for either the L^1 or L^2 norm.

A subset I of coordinates is represented by an object of the class `Coordinates`. Such a subset defines a projection L_I of the lattice. A `CoordinateSets` object can represent a set of such subsets of coordinates; i.e., a set of projections of the lattice. Examples of such sets of subsets are given in Eqs. (32) and (33).

In the subclasses of `Weights`, named `Weights`$\cdots$, facilities are offered to give different weights to specific subsets of coordinates (uniform weights, product weights, order-dependent weights, POD weights, projection-dependent weights). This can be used to compute FOMs as discussed in Section 10.

The following files provide basic tools used mostly in other classes of files. They are described later in this section. `EnumTypes` collects the definitions of all basic enumeration types used in Lattice Tester (some are also used in other packages). `FlexTypes` defines the flexible integer and real types for vectors and matrices. `Util` implements simple utility functions. `Num` implements some mathematical functions. `NTLWrap` extends certain NTL classes and offers a few basic utilities not offered in NTL. `Random` implements a 64-bit uniform random number generator, used when we make random selections. `ParamReader` provides functions to read data from a file. `Writer` and `WriterRes` provide functions to format and write output. Some of these functions are not (or no longer) used directly in Lattice Tester, but they may be used elsewhere.

11.2 Representing large numbers, vectors, and matrices

Floating-point real numbers. Lattice Tester uses flexible types to represent integers and real numbers. The generic types are named `Int` for the integers and `Real` for the real numbers. For the `Int` type, one may use 64-bit integers (`int64_t` in C, commonly named `long` in NTL and C++) when this is sufficient to represent all basis coordinates and make the required integer computations (but see also the next paragraph), or use `NTL::ZZ`, which can represent integers of arbitrary size. For the `Real` type, when `Int = ZZ`, we have the same four choices as in the LLL functions of NTL: `double`, `xdouble`, `quad_float`, and the `RR` arbitrary-precision floating-point type of NTL. When `Int = int64_t`, the only admissible choice is `Real = double`. This makes five combinations of types. The file `FlexTypes.h` defines the following code names for these five combinations:

code	Int	Real	details
LD	int64_t	double	
ZD	ZZ	double	
ZX	ZZ	xdouble	https://libntl.org/doc/xdouble.cpp.html
ZQ	ZZ	quad_float	https://libntl.org/doc/quad_float.cpp.html
ZR	ZZ	RR	https://libntl.org/doc/RR.cpp.html

In our experiments, we found that using LD was never much faster than using ZD, and sometimes it was even slower, depending on the compiler and computer. We now briefly describe the `xdouble`, `quad_float`, and `RR` floating-point types. There are more detailed explanations in the NTL files at the links given in the last column of the table.

An `xdouble` (extended double) has the same precision as a `double` (about 53 bits) but an extended exponent range, so much larger real numbers can be handled. It is represented by an ordinary `double` that contains the mantissa and an exponent, together with a 64-bit integer (`long`) that stores another number used to increase the exponent.

A `quad_float` (quadruple precision) gives about 106 bits of precision. The number x is represented as $x_1 + x_2$ where x_1 is a `double` that represents x with about 53 bits of precision as usual, and x_2 is another `double` used to add an extra 53 bits to the mantissa. The value of $|x_2|$ never exceeds the value represented by the least significant bit of x_1 if we put this bit at 1, so it does not change the mantissa that is already in x_1, except perhaps for its last bit.

The `NTL::RR` type refers to an arbitrary precision floating-point real number $x = 2^e \times m$, represented by (m, e), where m is a p-bit mantissa represented as a `ZZ` integer, and e is the exponent represented as a 64-bit integer (`long`). By default, the mantissa has $p = 150$ bits, but this p can be changed via the `RR::SetPrecision` static function. The current precision can be saved and restored by using a `RR::RRPush` object. The results of all operations with `RR` objects are rounded (after the operation) to p bits of precision, so the relative error does not exceed 2^{-p}.

A types code can be used in application programs to select the appropriate combination. For this, it suffices to define the variable `TYPES_CODE` to one of these four values, at the beginning of the program, before including the `FlexTypes.h` file and before any other instruction. See the program `TestBasisConstructSmall` in the examples. For instance, to select `Int = ZZ` and `Real = double` (a common choice), one would use the line:

```
#define TYPES_CODE ZD
```

Another way of selecting the two flexible types `Int` and `Real` is to pass the desired types in the class and function templates. The program `TestReducersSpeed` in the examples shows how to do that. In a nutshell, if a class or a function is declared as

```
template<typename Int, typename Real> fname(...);
```

calling it as in

```
fname<NTL::ZZ, double>(...);
```

will fix `Int` to `NTL::ZZ` and `Real` to `double` inside the function, and inside the functions called by this function with these template parameters, recursively. The compiler will then compile the code with the right types.

High-accuracy real numbers are needed mostly for the Cholesky decomposition in the BB procedure for computing a shortest vector, and also in the LLL and BKZ functions. Using `xdouble` or `quad_float` can accommodate larger or more accurate numbers, but makes the programs run more slowly. The `RR` type can handle arbitrarily large real numbers, but the operations are then much slower than for the other types. See the example `TestReducersSpeed` for speed comparisons.

In applications in which Lattice Tester is used to screen out a very large number of lattices to find a few good ones, one can first perform the computations with the standard type `double`, and then recompute (verify) with the higher-precision floating-point numbers only for the lattices that have been retained.

Vectors and matrices. Vectors and matrices of `Int` and `Real` are also defined in `FlexTypes.h`. We have `IntVec` and `IntMat` for integers, and `RealVec` and `RealMat` for real numbers. These are implemented as NTL vectors and matrices. These types are sometimes used in templates when instantiating objects.

The `NTL::vector` objects are allocated some space initially, then enlarged whenever needed. When they are resized to larger values, the space they occupy increases. When they are resized to smaller values, the space they occupy does not decrease, only their current dimensions (given by hidden local variables in the object) decreases. For a vector, `length()` returns the current (or logical) length (or size), while `MaxLength()` returns the space that the object occupies, which is at least as large as the largest length it had in its life. The function `SetLength(a)` sets the current length to `a` while `SetMaxLength(m)` sets the current max length (space) to `m`. By default, range checks for indices are not done, but if `NTL_RANGE_CHECK` is defined, code is added to test if `0 <= i < v.length()`. This check is not performed by default. See `https://libntl.org/doc/vector.cpp.html` for more details.

The `NTL::matrix` objects are implemented in a similar way, but with a key difference: when the number of columns change, the matrix is destroyed and a new matrix object is created (and initialized), which brings significant overhead. If only the number of rows is changed, the matrix is resized just like a vector. For this reason, we really want to avoid resizing the number of columns in those matrices. The function `SetDims(r, c)` changes the dimensions to r rows and c columns. The length of the rows cannot be larger than c as for vectors. If `mat` is a matrix, then `mat[i]` returns row i, which is an NTL vector, and `mat[i][j]` returns element (i, j) of the matrix.

To avoid the resizing of matrices, in Lattice Tester we often reserve space for the largest dimensions that we need and reuse the same objects without resizing them. For example,

each basis is stored in a `IntMat` object with `maxDim` rows and columns, which is allocated only once. If the current basis has `dim` $<$ `maxDim` dimensions, then only the upper-left corner of the matrix (first `dim` rows and columns) is used to store the current basis. Our functions have been implemented to use the matrices in that way. Each `IntLattice` object has internal `dim` and `maxDim` variables.

On the other hand, the LLL and BKZ implementations of NTL do not work that way. They take matrices that must have the exact right dimensions. For this reason, we have modified the files `LLL_FP`, `LLL_RR`, etc., from NTL to new versions named `LLL_FPInt`, `LLL_RR_lt`, etc., that can work with only parts of the matrices, as just described. The file `LLL_FPInt` also works for both `Int = ZZ` and `Int = int64_t`. The modified static functions also return the square lengths of the basis vectors (those in NTL do not).

The `TestMatrixCreationSpeed` example in Section 12.2 illustrates the importance of avoiding the frequent creation of new objects, e.g., creating new objects inside functions that are called several times, or resizing the number of columns of an `IntMat` object frequently.

11.3 IntLattice and IntLatticeExt

`IntLattice` is a base class for Lattice Tester. It represents an arbitrary integral lattice in t dimensions, with a primal basis made of t independent t-dimensional integral vectors, a scaling factor m, an m-dual basis, and a choice of norm (L^1 or L^2 norm). Various constructors are available. Usually, only the primal basis or only the m-dual basis is maintained, not both. The basis vectors are stored as the rows of a matrix of integers (`IntMat`) and their norms are stored in a vector of real numbers (`RealVec`). These norms are not always updated immediately when vectors are changed, but only when needed. There are several methods to set the primal or the m-dual basis to a given matrix, the norms of the primal or m-dual basis vectors, to check if the norm of a vector is up-to-date or not, to permute basis vectors, to sort them by length, to print the basis matrices, etc. The function `buildProjection` constructs the lattice that corresponds to the projection of the current `IntLattice` over a specific set of coordinates, and returns it in another `IntLattice` object passed as a parameter. The function `buildProjectionDual` does the same for the m-dual of this projection. These methods have default (general) implementations that are intended to be overridden by faster specialized implementations in subclasses. An `IntLattice` object contains several protected variables, so we should avoid creating many of these objects and rather reuse the same one, for example when searching for good lattices.

The class `IntLatticeExt` offers additional *virtual* methods which must be implemented in subclasses. The reason is that these methods can be implemented only in subclasses that construct specific types of lattices. They permit one to build a basis for the lattice, to build a basis for the projection of the lattice over a subset of coordinates, to extend the dimension of the lattice by one coordinate while leaving the current basis coordinates unchanged, to write a string that describes the lattice, and all of this for either the primal or the m-dual.

In the Modula-2 version, the primal and m-dual bases were maintained together, so they were always m-dual to each other. This is no longer true. They are now maintained almost

independently of each other, and only one of them is maintained, in particular because LLL, BKZ, and BB change either the primal or the dual, but not both. There is a current dimension for the primal basis and another one for the m-dual basis. When one of them is not built, its current dimension is 0. These dimensions are accessible via `getDim()` and `getDimDual()`.

11.4 BasisConstruction

This file offers *only static functions* to construct a basis from a set of generating vectors that are not necessarily independent, to construct a triangular basis, to construct the basis for a projection over a given subset of coordinates, and to compute the m-dual of a given basis. These functions use the algorithms discussed in Sections 5 and 6. The implementation uses `IntMat` matrices and some functions rely on NTL. Most of the computations on basis vectors in these functions are done modulo m (except for the vectors $m\boldsymbol{e}_i$ which cannot be replaced by $\mathbf{0}$), so no number should exceed m.

The functions `upperTriangularBasis` and `lowerTriangularBasis` construct an upper-triangular basis and a lower-triangular basis, respectively, using the gcd construction algorithm of Section 5.2. The function `LLLBasisConstruction` constructs an LLL-reduced basis from a set of generating vectors. The returned basis is not triangular in general, but it is usually comprised of shorter vectors. All these functions assume that the rescaled unit vectors $m\boldsymbol{e}_i$ always belong to the lattice, and they add them implicitly to the set of generating vectors.

The functions `mDualBasis`, `mDualLowerTriangular`, and `mDualUpperTriangular` compute the m-dual of a given basis for the general case and for the triangular case, respectively, as explained in Section 6. The first one is more general but it is much slower.

The function `projectMatrix` takes a given lattice basis and a given set I of coordinates, and extracts a set of generating vector for the projection of this lattice over the coordinates in I. The functions `projectionConstructionLLL` and `projectionConstructionUpperTri` use LLL and the GCD upper-triangular construction method, respectively, to compute a basis for the projection from the set of generating vectors. These methods are for arbitrary lattices and projections. In the subclasses of `IntLatticeExt`, there are more efficient specialized functions that exploit the structure of certain types of lattices to directly construct bases for projections of the lattice and for the m-duals of those projections. For example, `Rank1Lattice` implements the direct construction methods discussed in Section 5.4.

All functions in the `BasisConstruction` file are static, so there is no need to create an object to use them. We also avoid as much as possible to create new vectors and matrices inside these functions, and the physical size of the `IntMat` objects can be larger than what is used, so the user can re-use the same vectors and matrices over successive calls.

11.5 LLL and BKZ functions

The header files `LLL_FPInt.h` and `LLL_lt.h` specify the functions we use for LLL and BKZ. The versions defined in `LLL_lt.h` are just slightly modified versions of the NTL implementations for `Int = ZZ` combined with any of the four `Real` types. Their implementations are in four separate `LLL_*_lt.cc` files. The file `LLL_FPInt.h` contains implementations that work for `Real = double` and `Int = int64_t` or `Int = ZZ`. These different implementations are available by using the templates for `Int` and `Real` in `BasisConstruction` and `ReducerStatic`.

11.6 ReducerStatic

The file `ReducerStatic` offers static functions to reduce a given basis via LLL or BKZ, using the L^2 norm (only) to measure vector lengths. Their implementation uses functions from `LLL_FPInt.h` and `LLL_lt.h`, which are our modified versions of corresponding NTL functions declared in `https://github.com/u-u-h/NTL/blob/master/doc/LLL.txt`. In `LLL_FPInt`, the `Int` type can be either `int64_t` or `ZZ`, and the LLL algorithm is implemented using mostly "`double`'s" for the Gram-Schmidt orthogonalization. In the other variants, the `Int` type must be `ZZ`, and the `Real` type can be `double`, `quad_float`, `xdouble`, or `NTL::RR`.

11.7 ReducerBB

The class `ReducerBB` implements functions to find a shortest non-zero vector in a given lattice via the BB algorithm [47], with either the L^2 norm or the L^1 norm. These functions require that a lattice basis has already been constructed. We recommend to always apply the LLL or BKZ pre-reductions from `ReducerStatic` before invoking the BB algorithm. See the example `TestReducersSpeed` in Section 12.8 for more on that.

Our BB algorithm is based on a Cholesky decomposition of the Gram matrix, as described in Section 8.3. We also implemented the version based a lower-triangular basis described in Section 8.2, to make comparisons, but the latter leads to a slower BB algorithms, because it generally gives much wider bounds.

Our algorithms do not use the m-dual basis, in contrast to the method of [13]. They only use the rescaled primal basis. If we want to compute a shortest vector in the m-dual lattice, we must build a basis for m-dual, then dualize the lattice so the m-dual becomes the primal (`IntLattice::dualize` does that) and apply the desired reduction methods. All the subclasses of `IntLatticeExt` must have facilities to build a basis for either the primal or the m-dual, for the whole lattice or a projection.

The `shortestVector` function computes a shortest vector in the lattice, using the norm selected for this lattice (either L_2 or L_1), while `reductMinkowski` computes a Minkowski-reduced basis. Both use a recursive BB procedure and several internal variables (including vectors and matrices) in `ReducerBB`. For this reason, they have no static version. These

functions do not apply any pre-reduction by themselves. Before calling them, one should reduce the basis via an LLL or BKZ reduction, to reduce the size of the BB search.

To use `shortestVector` or `reductMinkowski`, one must create a `ReducerBB` object that maintains the internal variables, and points internally to an `IntLattice` object. Creating a new `Reducer` object for each `IntLattice` that we want to handle is inefficient and should be avoided. It is recommended to create a single `Reducer` object that contains an `IntLattice` object with a maximal dimension large enough to handle all the lattices that we want to examine, and just update the *contents* of this `IntLattice` object when going from one lattice to the next. The norm type, dimension, basis, vector lengths, etc., will be taken from this `IntLattice` object.

Changing the `IntLattice` object inside the `ReducerBB` can also be done easily and efficiently with the function `setIntLattice(&lat)`. In most cases, this function just changes the pointer to the new internal `IntLattice` object. If the reserved space for the internal vectors and matrices in the `ReducerBB` object is not sufficient for the dimension of the new `IntLattice`, this space is also enlarged as needed. It is never reduced. The other internal variables do not have to be updated when the internal lattice is changed, those that are needed are recomputed inside the `shortestVector` and `ReductMinkowski` functions.

In `ReducerBB`, the amount of details that is shown on the terminal during execution can be changed by the `setVerbosity` function. This can be useful in case we want to follow in details what transformations are made and what happens at each node of the BB tree.

11.8 Normalizer

The class `Normalizer` and its subclasses compute and store the normalizing constants discussed in Section 9. A specific `Normalizer` should be created by invoking the constructor of a subclass. The preferred constructor is the one that takes $\ln m$ and k as input. This constructor assumes that the primal lattice has been *rescaled by a factor* m, and that its density in t dimensions (after rescale) is $\tilde{\eta}_t = \eta_t/m^t = m^{k-t}$ for $t > k$ and 1 for $t \leq k$. This corresponds to the lattice L_t generated by an MRG with modulus m and order k, whose density *before the rescaling* is $\eta_t = m^k$ for $t > k$ and m^t for $t \leq k$. The bounds are computed for the *rescaled lattice* Λ_t, for which the log density is $\max(0, k-t)\ln m$, which depends on t. For the *m-dual lattice*, the density is $\eta_t^* = m^{-k}$ for $t \geq k$ and $\eta_t^* = m^{-t}$ for $t < k$, so the log density is $-\min(t, k)\ln m$. The corresponding bounds are computed by the constructor. By default, the bounds are computed for the L^2 norm, but they will be computed for the L^1 norm if that choice of norm is passed to the constructor. In that case, the constants γ_t are replaced internally by the modified $\gamma_t^{(1)}$ defined in Section 9. This is done in the function that returns the value of γ_t in the subclasses.

Passing $(\ln m, k)$ to the constructor of a subclass will give the bounds for the rescaled primal lattice, while passing $(-\ln m, k)$ will give the bounds for the m-dual lattice. The logs of the bounds for the rescaled primal and for its m-dual are given in (28) and (30). These bounds are computed internally by the function `computeBounds(logm,k)` for a selected range

of dimensions, usually for up to 48 dimensions only, because in most cases the estimates of γ_t are available only for $t \leq 48$. If we have passed $-\ln m$ for the first parameter, the function finds that this parameter is negative and computes the bound for the m-dual, otherwise it computes the bound for the rescaled primal. Then the exponential function is taken to recover the final bounds from their logs. We work with the logs of the bounds because the density itself is sometimes extremely large or extremely small. There is also another constructor that takes the log of the lattice density as input, but it can work only when the density is the same for all t, which is not true (at least for the primal lattice) in our setting.

The different subclasses use different estimates of the constants γ_t, so they give slightly different bounds. Some choices of estimates are *lower bounds* on the constants γ_t, so they do not provide true upper bounds on the shortest vector lengths, but only low-biased estimates. Other choices are *upper bounds* on the γ_t. The bounds are precomputed for a selected range of dimensions t by the constructor. They can then be accessed either for one dimension at a time (via the `getBound` function) or as a vector for all the selected dimensions (via `getBounds`). In most cases, the maximum dimension is 48. The available subclasses of `Normalizer` are the following:

class name	bound type	norm	symbol	origin
`NormaBestLat`	lower	L^2	γ_t^{B}	Best known lattice packing [8]
`NormaLaminated`	lower	L^2	γ_t^{L}	Best laminated lattice [8]
`NormaMinkHlaw`	lower	L^2	γ_t^{Z}	Minkowski-Hlawka lower bound
`NormaBestUpBound`	upper	L^2	γ_t^{C}	Best known upper bound [7, 6]
`NormaRogers`	upper	L^2	γ_t^{R}	Roger's upper bound [8]
`NormaMinkL1`	upper	L^1	γ_t^{M}	Upper bound from Minkowski [67]

We recommend using either the best lower bound given in `NormaBestLat` or the best upper bound given in `NormaBestUpBound`. Both use the exact values of γ_t when they are known. Otherwise, the latter use the bounds from [7] for up to $t = 36$ dimensions, and the bounds of Rogers for $t > 36$. The best lower and upper bounds for $t \leq 36$ are compared in [7, 6] in terms of bounds on sphere packing densities. For the L^1 norm, one can use modified versions of these bounds by transforming them into bounds on $\gamma_t^{(1)}$, as explained in Section 9. This can be done simply by passing the appropriate norm to the constructor of the `Normalizer` object.

When making a search and examining millions of lattices, one should create a single `Normalizer` object and re-use it, not construct a new one for each lattice that is examined.

11.9 Weights

The abstract class `Weights` provides an interface to specify *weights* ω_I given to projections I when defining a figure of merit as in (31), or more generally as in [51], Equation (8). Its subclasses (whose names start by `Weights...`) define different types of weights, as explained in Section 10. The class names are the names of the weights. These different types of weights

are used in LatNet Builder. When studying RNGs, it is most common to use either uniform weights (all weights are 1) or order-dependent weights (the weight depends only on the cardinality of the projection). When using (31) with a large number of low-dimensional projections, for example, we may want to give smaller weights in the dimensions in which the projections are too numerous, because otherwise the minimum is almost always attained for one of them. Since the FOM is a minimum and we divide by the weight, a smaller weight gives less importance to the corresponding projection in the FOM.

To apply such weights to projections, one should create a arbitrary `Weights` object, specify the desired weights inside that object, and then call the function `getWeight(projection)` on that object each time we want to retrieve the weight of a given "`projection`".

11.10 Coordinates and CoordinateSets

A `Coordinates` object is a C++ set that represents a set of coordinate indices, often non-successive, and which determine a projection. These objects are created and returned by the classes in the `CoordinateSets` namespace. The coordinates are assumed to start at 1 (the first coordinate) and be listed in increasing order in the set.

A `CoordinateSets` object corresponds to a set of subsets of coordinate indices, i.e., a set of `Coordinates` objects. For example, the set $S_s(t_s)$ in (33) corresponds to a `CoordinateSets` object. Its alternative $S_s^{(1)}(t_s)$ in (35) is another one. The sets

$$\bigcup_{2\le s\le d} S_s(t_s) \qquad \text{and} \qquad \bigcup_{2\le s\le d} S_s^{(1)}(t_s)$$

are also `CoordinateSets` objects, which can be constructed using the functions provided in the class `FromRanges` in `CoordinateSets.h`. These types of objects are used when computing a FOM such as (31). The function `FigureOfMeritM::setTVector` shows how to construct them. The idea is to create a `FromRanges` object, then add projections to it. In the following example, `projSet` represents a set of projections. The two `includeOrder` statements add the projections in $S_2(8)$ and $S_3(8)$, namely all pairs and triples of distinct coordinates from $\{1,\dots,8\}$.

```
CoordinateSets::FromRanges projSet = new CoordinateSets::FromRanges;
projSet->includeOrder (2, 1, 8, false);
projSet->includeOrder (3, 1, 8, false);
```

The "`false`" in these statements indicates that coordinate 1 is not always included. To include $S^{(1)}(2,8)$ and $S^{(1)}(3,8)$, i.e., the subsets of the pairs and triples that contain coordinate 1 in the sets above, one can do

```
projSet->includeOrder (2, 2, 8, true);
projSet->includeOrder (3, 2, 8, true);
```

The "`true`" indicates that we always include coordinate 1. The second parameter is 2 because coordinate 1 is already included, so the other coordinates must be from the set $\{2,\dots,8\}$.

11.11 Figures of merit

The classes whose names start by `FigureOfMerit` compute figures of merit (FOMs) for an arbitrary `IntLatticeExt` object. In particular, `FigureOfMeritM` and `FigureOfMeritQ` compute the FOMs defined in (31) and (37) for the rescaled primal lattice and its projections, whereas `FigureOfMeritDualM` does it for the m-duals of the projections. Other FOMs may be added in the future.

When computing an FOM, the lengths of the shortest vectors in the projections can be just approximated by the lengths of the shortest basis vector obtained after applying pre-reductions such as LLL or BKZ, or they can be calculated exactly by using the BB algorithm after applying these pre-reduction. The former is faster but not exact.

The constructor of `FigureOfMeritM` requires the integer vector $(t_1, \ldots, t_d)$ used in (31), a `Weights` object that defines the weights given to the projections, a `Normalizer` object used to normalize the vector lengths in the FOM (see Section 11.8), an optional `ReducerBB` object used in case we perform the BB (if no `ReducerBB` is given, no BB is performed), and an optional `includeFirst` boolean parameter which must be set to "`true`" if we want to include only the projections that contain coordinate 1 as in (35) (the default value is "`false`"). The reductions are always applied in the order LLL $\rightarrow$ BKZ $\rightarrow$ BB. Each of them can be "on" or "off", which gives 8 possible combinations. By default, the pre-reduction method is BKZ with $\delta = 0.99999$ and `blocksize = 10`. This can be changed via the functions `setLLL` and `setBKZ`. To remove LLL or BKZ, it suffices to set its δ parameter to 0.0. After a `FigureOfMeritM` object has been created, one can still change its vector $(t_1, \ldots, t_d)$, its weights, or its normalizer, via the appropriate functions.

The class `FigureOfMeritMDual` is very similar to `FigureOfMeritM`, except that it computes the shortest vectors in the m-duals of the projections. As we saw earlier, *this is not the same* as computing them for the projections of the m-dual, which is what would be done if we first dualize the full primal lattice and then use `FigureOfMeritM` for the dual.

In either case, the function `computeMerit` computes the FOM for a given lattice. It has two parameters: the lattice for which we want to compute the FOM, and a second `IntLattice` object used internally to store the projections over non-successive coordinates. The `maxDim` (dimension of the array that stores the basis) for this second object should be large enough to store the largest projection. We recommend to re-use the same one when the FOM is computed for many lattices, and this is the reason why we put it as a parameter.

The norm used to compute the vector lengths is the norm associated with the lattice for which we compute the FOM. The norm used for the normalization is the one associated with the `Normalizer` object that was passed to the constructor. These two choices of norm should be the same, otherwise the normalization will be inconsistent. Likewise, the values of `(m, k)` for the given lattice must be consistent with those that were used when computing the bounds in the `Normalizer`.

The `computeMerit` function first calls `computeMeritNonSucc`, which goes through all projections in the sets $S_2(t_2), \ldots, S_d(t_d)$ in this order, then it calls `computeMeritSucc`, which

goes through the projections in $S_1(t_1)$. These two functions can also be called directly if desired. For each projection, the function finds (or approximates) a shortest nonzero vector, computes its square length, computes the normalized merit for that projection, which is usually a value between 0 and 1, and it updates the minimum. Each of these functions returns the minimum merit that was found. They have an optional input parameter "`minmerit`" in case we want to pass a current minimum value obtained by looking at previous projections. By default, this value is "infinite" (`MAX_DBL`).

These functions use a minimal threshold of acceptability of the FOM named `lowbound`. As soon as the normalized merit value of one of the projections gets below `lowbound`, we know that the FOM will be below this bound and we can immediately stop computations for this lattice. By default, this lower bound is zero, but it can be changed via the function `setLowBound`. This "early exit" can make a huge difference in the efficiency when we examine thousands or millions of candidates to search for good lattices. The `lowbound` may be set to the best FOM found so far, for example.

The level of details that is printed on the terminal or saved in local variables when the figures of merit are computed can be set by the `setVerbosity` and `setCollectLevel` functions. These functions take an integer from 0 to 4. With the default value (0), nothing is printed of saved. With values from 1 to 4, we print or collect increasingly more details, such as the worst-case projection, the corresponding shortest vector, its square length, its merit value, etc.

11.12 Rank1Lattice

This is a subclass of `IntLatticeExt` to handle rank-1 lattices as described in Section 5.4. This class permits one to test the other facilities by providing a simple concrete implementation of the virtual class `IntLatticeExt`. It is used for most of the examples of Section 12. To construct a `Rank1Lattice`, we need to select a modulus m and a generating vector $\boldsymbol{a} = (a_1, \ldots, a_t)$ in t dimensions, with $a_1 = 1$. For a Korobov lattice, this vector will be $\boldsymbol{a} = (1, a, a^2 \bmod m, \ldots, a^{t-1} \bmod m)$ and it suffices to select the integer a. Otherwise, we pass the vector $\boldsymbol{a}$ and the condition that $a_1 = 1$ is tested internally. It is also assumed that $\gcd(a_j, m) = 1$ for all j, although this is not tested. The functions described in Section 5.4 are used to construct bases for the lattice, for its projections, and for their m-duals.

11.13 EnumTypes

This file collects the definitions of various enumeration types used in Lattice Tester or in other packages that depend on Lattice Tester. Some of them may be no longer used.

11.14 Other files

The following files provide basic and general utilities that can be used either in Lattice Tester and in other packages that depend on Lattice Tester. Many of these functions might not be used anywhere anymore after all those years of changes in these packages, but they are still available in case they could be useful.

`Util` implements basic utility functions to make conversions across different types, reset variables, compute power functions, square roots, logarithms, inverses, integer divisions, rounding, modulo operations, Euclidean algorithm for gcd, scalar product, norms, vector and matrix operations with our flexible types, some streaming operators, etc. Some of these functions rely on NTL, or implement overloads of NTL functions for standard types, so these functions can be used with our flexible types.

`Num` implements mathematical functions such as factorial, Bernoulli polynomials, harmonic functions, and Fourier series. They are not used directly by Lattice Tester, but by other packages that depend on Lattice Tester.

`NTLWrap` offers a few basic utilities not available in NTL. It is in the NTL namespace, because it can be seen as an expansion of NTL.

`Random` implements a 64-bit uniform random number generator, used when we make random selections, for example when searching for generators or QMC point sets with a good lattice structure.

`Chrono` can provide multiple "stopwatch" objects to measure the CPU time consumed by various parts of a program, and return the results in readable string format, in the time units of our choice. `Chrono` is often more convenient than `ctime` because it predefines various output formats, so it is more flexible.

12 Examples of programs that use Lattice Tester

Here we discuss testing programs that use Lattice Tester and are included in the GitHub distribution. Some of them were used to make experiments to compare algorithms and implementations for certain tasks. The descriptions below should be read while looking at the code and result files for these examples. These files are available with the distribution in the `examples` and `examples/results` subdirectories and we do not reproduce them here, but we show and discuss subsets of the results. [4]

[4] From Pierre: * In most of these examples, we create a vector `sqlen` of size 1 to collect the square shortest vector length. There would be no need to do this if the reducer and BB functions would return this length either in a `Real` or in a `double`, and return 0 when the function fails. This would be a little simpler, although we would also have to collect the value in a `Real` or in a `double`. Sometimes, the square length might not fit in a `double`, so better use a `Real`. Note that currently, the reducers return the dimension of the reduced basis, and the BB returns only a boolean. Making this changes will trigger *a lot* of small changes all around, in the code, in the doc, in the examples, in the guide, etc., so we have to look carefully at all the consequences before making it. Not urgent.

We will use the notation LLLx to denote LLL with factor $\delta = 0.x$, BKZx-k for BKZ with factor $\delta = 0.x$ and blocksize k, and "`+BB`" to indicate that BB was applied.

12.1 TestBasisConstructSmall

This small example illustrates the use of static functions from the file `BasisConstruction.h`. These functions apply directly to `IntMat` objects that contain the basis matrices rather than to `IntLattice` objects. We examine a five-dimensional lattice obtained from an LCG with a small modulus $m = 1021$ and multiplier $a = 12$. We also look at its projection on a subset of three coordinates.

In the program, the flexible types `Int` and `Real` are selected and fixed once for all in the first line of code by uncommenting one of the five choices of combination. For example, we uncomment "`#define TYPES_CODE ZD`" to select `Int = ZZ` and `Real = double`. If we change this selection, we have to recompile the program. In the next examples, we will show how to make the selection by using template parameters and make several selections of types from the same code, without recompiling.

We then declare and resize `IntMat` objects to hold the bases that we will manipulate, and a vector `sqlen` of size 1 that we will use to recover the square length of the shortest basis vector after each LLL reduction. The constructor of `Rank1Lattice` constructs a `IntLatticeExt` object in which the primal basis will be maintained. The function `buildBasis` constructs an initial upper-triangular basis directly, as described in Section 5.4, and we copy it in `basis1`. This basis and the length of its first basis vector are printed here and also between the following function calls, to show what is going on. The reader should execute the program and look at the code and the output while reading the following.

We apply LLL with `delta = 0.5` to obtain a basis which is not triangular but is made of shorter vectors. We do it again with `delta = 0.99999`, for comparison. In both cases, the square length of the first basis vector after LLL is 34190. Then we call `lowerTriangularBasis` to transform this reduced basis to a lower-triangular basis in `basis2`, and we call `upperTriangularBasis` to transform the latter into an upper triangular basis, which differs from the initial one in only one entry: -314 instead of 707, which is equivalent modulo 1021. We then compute the m-dual of this upper-triangular basis, and reduce it by applying LLL with $\delta = 0.99999$. The shortest vector in that m-dual basis has square length 6. All these bases are maintained directly as `IntMat` objects, not inside the `Rank1Lattice` object.

After that, we look at the projection of this lattice over the subset of coordinates $\{1, 3, 5\}$. We first construct a `proj` object that contains these three coordinates. We then show how to compute a basis for this projection in three ways: (1) we put in `basisProj` a set of generating vectors for this projection and we apply an LLL construction to this set to obtain a basis in `basisProj`; (2) we make an upper-triangular projection construction from `basis2` into `basisProj`; (3) we create a new `projLattice2` lattice object and we call the function

`buildProjection` to build a basis for the projection in this object. All these bases are three-dimensional, even though they are sometimes stored in higher-dimensional matrices. Using LLL may be appropriate only when we only want a basis for the primal projection and want it to be reduced. The upper-triangular method is faster and more appropriate if we also want a basis for the m-dual of the projection, as shown a few lines below. These first two methods work for an arbitrary lattice basis. The third method has a specific implementation which exploits the fact that for a `Rank1Lattice`, under mild conditions, the basis of a projection and its m-dual can be constructed directly, as explained in Section 5.4. It is usually more efficient for this reason.

Our next step is to pass the triangular basis `basisProj` to `mDualUpperTriangular` to obtain the m-dual basis `basisDualproj`. We then use LLL to reduce this m-dual basis and we look at the shortest basis vector. A basis for the m-dual of the projection can also be constructed directly by using the specialized function `buildProjectionDual` in `Rank1Lattice`, which is typically faster. We show how to do that.

Finally, we show what happens when we construct a basis for the projection of the m-dual basis for the full lattice instead of the m-dual of the projection. In the basis that we get for the projection of the m-dual, all three basis vectors have length 1. This illustrates the fact that projecting the m-dual directly does not give the same lattice than taking the m-dual of the projection.

The program output is in file `examples/results/testBCSmall.res`. This program can be run with any of the five different types codes, and with various choices of the modulus m and multiplier a. The reader is encouraged to experiment with it.

12.2 TestMatrixCreationSpeed

This example compares different resizing options for an `IntMat` object when the required matrix size changes frequently. A first option is to create a single matrix of sufficiently large dimension for all the tasks that we need to do, and use this matrix without ever resizing it. When the required space is smaller, we just use the upper left corner of that matrix, for the size we need. This way, we avoid reserving blocks of memory for new matrices again and again. One potential drawback with this approach is that if we need a very large matrix once in a while and only a small matrix most of the time, using the very large matrix all the time may slow down the program by clogging the cache memory more than necessary.

A second (intermediate) option is to create a single matrix object as above, with a fixed number of columns large enough for all our tasks, and resize the number of rows when needed. That way, when the number of rows is reduced, the same block of memory is kept, as explained in Section 11.2, and a new block of memory is reserved only when we need more than what is currently allocated. This can be advantageous compared with the first option in situations in which we need a small matrix for a long time and a larger matrix only later.

A third option is to resize the single matrix each time the required dimension changes. If it changes only once in a while, this will not bring much overhead, and may be advantageous compared to the first two options because we do not have to always carry a large matrix.

Two other options (four and five) are to always resize the matrix or to always create a new matrix when we need one. These two options are practically equivalent and can bring significant overhead if we need new matrices very often.

The program `TestMatrixCreationSpeed` compares these five options in two different settings, using the six matrix sizes $s \in \{5, 10, 20, 30, 40, 50\}$. In the first setting, for each of the five options, for each matrix size s, we repeat the following $r = 100,000$ times: we change the first s diagonal elements of our `IntMat` object, whose dimension must be at least $s \times s$. In replication i, we add $i + j$ to the element (j, j). In the end, we add the diagonal elements for each size and we print the sums, to make sure that the compiler does not optimize out the additions. The sums are not the same for all five options because some reset the matrices differently. In the second setting, we change the order of the loops: The outside loop is on the replication number and the inside loop is on the size s. Option will resize the matrix much more eften in this case. These two settings are implemented by the functions `testLoopDimOut` and `testLoopDimIn`. The results are in `testMCSpeed64` for `Int = int64_t` and in `testMCSpeedZZ` for `Int = NTL::ZZ`. The timings are summarized in Table 3.

As expected, everything takes more time with the `ZZ` integers than with `int64_t`, by a factor of about 3 to 5. Options 4 and 5, as well as option 3 in the second setting, are *much* slower than the others. They are the cases where we reallocate a new block of memory (via a resize or a new object creation) each time we use a matrix. For the other methods, for which the resize occurs rarely or never, the speeds are comparable. Bottom line: it is worthwhile to avoid reallocating memory for matrix objects. Note that reading the clock for the timings brings overhead that could be significant when we do fewer replications or if we do a separate timing for each dimension. Instructions that were doing that are commented out in the program.

Table 3: Timings (microseconds) for the `TestMatrixCreationSpeed` example.

Method	`Int = int64_t`		`Int = NTL::ZZ`	
	setting 1	setting 2	setting 1	setting 2
1. One matrix, no resize	49653	66500	284391	281340
2. One matrix, resize rows often	58172	58178	292091	294787
3. One matrix, resize when needed	52884	2654279	284623	7638361
4. One matrix, resize often	2682655	2706314	7908641	7607095
5. New matrix each time	2413822	2393043	8556413	8495962

12.3 TestBBSmall

This example compares different ways of finding a shortest vector in a lattice by using `shortestVector` from `ReducerBB` to apply the BB procedure. We consider the L^1 and L^2 norms, compare the Cholesky and Triangular decompositions in the BB algorithm, and do this for both the primal and m-dual lattices. This program was used for the examples of Section 8.7.

The function `findShortest` finds a shortest vector for a given setting. This function is given a Korobov lattice `korlat` and a reducer `red`. It first builds a lattice basis, applies LLL to get a reduced basis and a first candidate for a shortest vector, whose square length will be used in the BB algorithm, and finally calls `red.shortestVector` to find a shortest vector. The latter function will print the shortest vector found and its square length since the verbosity level of the reducer was set to 2.

The detailed output for $m = 1021$ with $a = 73$, and for $m = 1048573$ with $a = 29873$, is in files `examples/results/testBBSmall10-4dim.res`, `testBBSmall10-8dim.res`, and `testBBSmall20-4dim.res`. Part of it is reproduced in Section 8.7.

12.4 TestBasisConstructSpeedLLL

This example illustrates again the use of functions from `BasisConstruction` and compares their timings, for LLL reductions in up to 70 dimensions. Here, we use template parameters to select the flexible types from the `main` function instead of fixing them at the beginning of the file. This way, we can run the program with all five combinations of types in the same run, without recompiling. In the `main`, we call `testLoop` with different choices for the two template parameters `<Int, Real>`. This function calls `transformBasesLLL` several times with the same template parameters to make timing experiments.

The function `transformBasesLLL` starts with an initial (primal) basis `basis0` for a Korobov lattice (that corresponds to an LCG), makes a copy of `basis0` into `basis1`, and applies successive reductions to `basis1`. It first applies LLL with $\delta = 0.5$ and looks at the squared length of the shortest basis vector, then continues with this reduced basis and applies LLL with $\delta = 0.8$, then continues again with $\delta = 0.99$, and finally with $\delta = 0.99999$, each time recording the squared length of the shortest basis vector. The values are labeled by LLL5, LLL8, LLL99, and LLL99999 in the results. After that, it restarts from the initial `basis0` and applies LLL directly with $\delta = 0.99999$, to compare the timings between the previous incremental reduction with increasing values of δ and a direct LLL reduction with δ very close to 1. This direct reduction is labeled LLL99999-new in the results. We want to see if the incremental reduction could be faster, and/or could lead to a shorter shortest vector, than the direct reduction. Then, the function calls `upperTriangularBasis` to recover an upper-triangular basis from the LLL-reduced basis, and `mDualUpperTriangular` to compute the m-dual of this triangular basis, which turns out to be lower triangular. It then applies the same sequence of incremental LLL reductions to this m-dual basis, as well as the direct reduction, with the same values of δ.

The `transformBasesLLL` function is called several times, as follows. We consider LCGs with modulus m and `numRep = 1000` different values for the multiplier a. For each value of a and each number of dimensions in $\{5, 10, 20, 30, 40, 50, 60, 70\}$, we construct the lattice basis `basis0` and call `transformBasesLLL`. We add the CPU times and average the square vector lengths over all values of a for each operation type (or method) and each number of dimensions. The timings are to compare the speeds. The average square vector lengths are used to compare the reduction methods in terms of the shortest vector lengths that are obtained, and to test for consistency and correctness. These sums should be the same when we change the `Real` type, for example. We ran this experiment for $m = 1048573$, a prime number near 2^{20}, and $m = 1099511627791$, a prime number near 2^{40}. In each case, we tried different combinations for the types `Int` and `Real`.

```
TestBasisConstructSpeedLLL with m = 1099511627791
Types: Int = NTL::ZZ, Real = double
Number of replications (different multipliers a): 1000

Timings for different methods, in basic clock units (microseconds):
Dimension:       10        20        30        40        50        60        70
LLL5          64869    219722    403349    662631    992828   1425139   1952965
LLL8           9454    105528    320263    558927    867797   1249442   1740953
LLL99          6439     59711    324137    756823   1158413   1640169   2256611
LLL99999       4446     22733     87235    235154    217145    341597    525429
LLL99999-pnew 86986    664243   1867248   3270545   4839269   6671719   8795632
UppTri         7345     19610     38004     60208     78052     96182    116560
mDualUT        2173     10891     31267     69461    130790    221707    345579
LLL5-dual     38815    106590    145790    191292    248674    321531    414551
LLL8-dual      8910     88210    214656    285721    365275    465224    585702
LLL99-dual     6054     53389    250845    477284    605111    750198    922590
LLL99999-dual  4281     22005     72351    160969    253837    364170    474355
LLL99999-dnew 46986    258713    646636    937362   1079113   1239884   1424416

Sums of square lengths of shortest basis vectors (same values for all flexible types):
Dimension:         20            30            40            50            60
LLL5          2.16230e+23   9.03316e+23   1.2074e+24    1.20893e+24   1.20893e+24
LLL99999      1.13522e+23   4.21957e+23   9.4600e+23    1.20851e+24   1.20893e+24
LLL99999-pnew 1.13838e+23   4.28475e+23   9.7951e+23    1.20849e+24   1.20893e+24
LLL5-dual        46.535        39.233        35.574        33.287        31.688
LLL8-dual        26.056        17.529        16.349        15.647        15.085
LLL99999-dual    24.939        14.647        12.784        12.226        11.873
LLL99999-dnew    25.152        14.923        13.271        12.819        12.487
```

Figure 1: Partial output of `TestBasisConstructSpeedLLL` with $m = 1099511627791$.

The complete results are in the directory `examples/results`, in `testBCSpeedLLL20.res` for the smaller m and `testBCSpeedLLL40.res` for the larger m, for the different types codes. Figure 1 shows part of the results for $m = 1099511627791$ with the `<ZZ, double>` types. Table 4 gives the total time to run `testLoop` for all the dimensions, with the different types codes. We see that for this example, in the computing environment we used, using the types

<ZZ, RR> is about 25 times slower than <ZZ, double>. We also find that using <int64_t, double> is *slower* than <ZZ, double>, because the LLL implementation is slower.

By comparing the sums of square lengths in Figure 1, we see that taking δ closer to 1 usually gives shorter vectors. This is clearly thee case in the m-dual lattice. For the primal, this is true in less than 50 dimensions, but after that the difference is much smaller. We also find that performing LLL by increasing δ incrementally can be faster than doing it directly with a δ very close to 1, and often leads to shorter vectors than when we call LLL directly with the largest δ. This suggests that an incremental approach may provide a better pre-reduction, which could lead to a faster BB algorithm afterward. As an illustration, in Table 1, in 30 dimensions, the sum of times for LLL5, LLL8, LLL99, and LLL99999 is about 1.13 seconds and the sum of square lengths of shortest vectors after these reductions is about 4.219×10^{26}, whereas the corresponding values for LLL99999-new are 1.87 seconds and 4.285×10^{26}. For the m-dual lattice, the numbers are about 0.58 seconds with 14647 for the square lengths in the incremental case and 0.65 seconds with 14923 in the direct case. The behavior is similar in larger dimensions.

Table 4: Total times in seconds to run the `TestBasisConstructSpeedLLL` example for up to 70 dimensions.

Code	<Int, Real>	$m = 1048573$	$m = 1099511627791$
LD	<long, double>	26.6	—
ZD	<ZZ, double>	18.5	60.5
ZX	<ZZ, xdouble>	85.2	320.3
ZQ	<ZZ, quad_float>	110.7	543.2
ZR	<ZZ, RR>	805.8	2363.3

The timings in the figure also suggest that `upperTriangularBasis` is much faster than LLL with $\delta = 0.5$ to compute a basis. This is when the number of generating vectors is already equal to the dimension. Note that for a `Rank1Lattice`, the basis can also be constructed directly as explained in Section 5.4.

12.5 TestBasisConstructSpeedTri

Here we make similar experiments as in the previous example, but we focus on comparing efficiencies of triangular and m-dual basis constructions methods instead of LLL. We start from an LLL-reduced basis of a rank-1 lattice that comes from an LCG, and we compare three ways of obtaining an m-dual basis: (1) via the general `mDualBasis` function, (2) by inverting a lower-triangular basis, and (3) by inverting an upper-triangular basis. In each case, we look at the time required to compute the m-dual, the time required to apply LLL to this m-dual basis, and the square length of the shortest vector obtained after LLL, to check if these shortest lengths are about the same for the three methods.

```
TestBasisConstructSpeedTri with m = 1099511627791
Number of replications (different multipliers a): 1000
Types: Int = NTL::ZZ, Real = double, LLL with delta = 0.5

Dimension:          10        20        30        40        50        60        70
Timings for the different tasks, in basic clock units (microseconds):
LLLPrimal        67535    224808    406621    669468   1008176   1441399   1986495
mDualBasis      107525    606000   1586903   3114663   5357904   8735251  13112882
LLLDualmDual      7719     41978     91636    141429    215628    371684    639091
LowTriP           8074     21428     33671     46946     61583     77690     94271
mDualLow          2447     11505     36044     81561    150934    249679    379418
LLLDualUT        41166    110638    153111    205156    273312    367328    486884
UppTriP           7640     20836     34103     46840     61152     77320     94629
UppTriP2          3376      7400     12982     20260     29104     38767     49751
mDualUp           2326     11096     34141     78909    150710    255054    397538
LLLDualLT        37877    102498    141079    188595    249021    327039    427511
LowTriDual       27095     81832    117833    160245    211429    270323    336945
UppTriDual       24677    102941    227294    385100    573943    792551   1034139
UppTriDualOld    75321    361897    820844   1496258   2443413   3719883   5352611
UppTriDual2      10875     33301     62133     98969    148263    206553    268886

Sum of squares after each of the LLL in dual, with delta = 0.5:
LLLDualmDual    227899     31490     27101     27413     26336     25707     24705
LLLDualUT       240555     46798     40303     35597     34554     31690     30720
LLLDualLT       240824     46535     39233     35574     33287     31688     30531

Sum of squares after each of the LLL in dual, with delta = 0.99999 :
LLLDualmDual    225731     25183     14521     12620     12885     13605     14358
LLLDualUT       225646     25035     14714     12992     12555     12299     12079
LLLDualLT       225642     25152     14923     13271     12819     12487     12266
```

Figure 2: Partial output of `TestBasisConstructSpeedTri` with $m = 1099511627791$.

Specifically, we evaluate the time taken by each of the following operations as implemented in the function `triangularBases`. This function starts with the initial (primal) basis `basis0` and applies LLL to `basis0` to reduce it. For each of the three methods, it will make a copy of `basis0` and work only with that copy, because the triangulation functions damage the basis on which they act. It first calls the `mDualBasis` function with `basis4`, a copy of `basis0` made with the exact correct dimensions because `mDualBasis` requires that, and recovers the m-dual in `basis5`, to which LLL is then applied. For the second method, `basis0` is copied to `basis1`, which is used to construct a lower-triangular basis in `basis2`, from which an upper-triangular m-dual basis is computed into `basisdual` as explained in Section 5, and LLL is applied to this m-dual basis. For the third method, `basis0` is copied again to `basis1`, which is used to construct an upper-triangular basis in `basis2`. To see how much faster the algorithm runs when the basis is already upper-triangular, it applies it again to `basis2` to obtain `basis1` (this is `UppTri2`). Then, the m-dual of `basis1` is computed into `basisdual` and LLL is applied to this lower-triangular m-dual basis.

After these three methods have been tried, the function compares triangulations in the

```
TestBasisConstructSpeedTri with m = 1048573
Number of replications (different multipliers a): 1000
Types: Int = NTL::ZZ, Real = double, LLL with delta = 0.5

Dimension:      10        20        30        40        50        60        70
Timings for the different tasks, in basic clock units (microseconds):
LLLPrimal      30630     75784    126755    194028    272800    386172    504242
mDualBasis     68956    296381    695048   1375940   2401785   3763748   5822523
LLLDualmDual    7721     34422     63813    105717    188594    339405    572687
LowTriP         7792     18358     29108     41242     54977     69851     86566
mDualLow        2362     11975     37416     82247    150454    247439    373449
LLLDualUT      24955     55120     82514    123192    177130    254118    356616
UppTriP         7617     18854     29270     41396     54973     69536     85109
UppTriP2        3501      8000     13992     21843     31244     41680     53267
mDualUp         2302     11681     36955     83995    159291    267457    414301
LLLDualLT      24373     53809     78673    115812    166293    232287    321837
LowTriDual     23929     55221     86816    127242    175466    230325    293517
UppTriDual     23514     94215    201257    331340    491562    671524    873623
UppTriDualOld  62418    275570    651031   1245900   2112957   3295329   4853358
UppTriDual2    10981     32576     61241     98267    143325    213744    277215

Sum of squares after each of the LLL in dual, with delta = 0.5:
LLLDualmDual   15453      7359      6735      6361      6027      5867      5708
LLLDualUT      15928      8649      7658      7066      6858      6476      6312
LLLDualLT      16137      9018      7967      7404      7083      6801      6593

Types: Int = int64_t, Real = double, LLL with delta = 0.5:
Timings for the different tasks, in basic clock units (microseconds):
LLLPrimal      20577     60583    107096    174427    255014    367332    501161
LowTriP         1107      2531      4206      6888      9689     13164     17133
mDualLow         471      1523      3551      6905     12335     19991     30542
LLLDualUT      23084     85694    173690    315645    530782    840609   1246320
UppTriP         1025      2306      4154      6462      9255     12399     16162
UppTriP2         566      1537      3131      5278      7875     11081     14723
mDualUp          425      1329      3208      6377     10939     17331     25673
LLLDualLT      19359     69813    142780    259480    433359    679195   1009453
LowTriDual      2678      5900      9326     14353     21119     28964     38609
UppTriDual      3049      9625     18664     30502     44744     62162     82340
UppTriDualOld   3486     16463     45618     98687    182035    302210    466088
UppTriDual2     1674      4564      8919     15185     22946     32813     44195
```

Figure 3: Partial output of `TestBasisConstructSpeedTri` with $m = 1048573$.

m-dual, by using a copy of `basisdual` each time. It first copies `basisdual` to `basis1` and transforms it to a lower-triangular basis in `basis2`. It copies `basisdual` to `basis1` again and transforms it to an upper-triangular basis in `basis2`. It repeats the latter with the old triangularization method of [10] instead, to compare. Finally, it calls `upperTriangularBasis` again with the already upper-triangular `basis2`, to see how much faster it will run.

The results are in files `testBCSpeedTri*.res`. Figure 2 shows partial results for $m = 1099511627791$ with the `<ZZ, double>` types and $\delta = 0.5$ for LLL. A first important observation from these results is that to obtain an m-dual basis, it is much faster to build a triangular basis and invert it (Methods 2 and 3) than to use the general `mDualBasis` function (Method 1). Methods 2 and 3 take about the same time. For the primal lattices, constructing a triangular basis is also much faster than calling LLL even with $\delta = 0.5$. For comparison, with $\delta = 0.99999$ the time required for `LLLPrimal` (not shown in the figure) is about five times more. On the other hand, when we look at the sums of squares in the lower part of the figure, we see that the shortest vector found by LLL is much shorter on average with $\delta = 0.99999$ than with $\delta = 0.5$, except in small dimensions. We also see that with $\delta = 0.5$, the shortest vector obtained by LLL is significantly shorter when the m-dual basis is constructed by Method 1 compared with Methods 2 and 3, whereas with $\delta = 0.99999$, we see the opposite. We do not have an explanation for this behavior. [5] To test if this could be caused by incorrect m-dual bases, we also applied the full BB algorithm after LLL to obtain the exact shortest vector in each case, and found that the three methods gave identical sums of squares (smaller than those in the figure) in that case.

Once we have a partially reduced m-dual basis, we observe that transforming it to a triangular one takes much more time than in the primal, and that computing an upper-triangular one is much slower than a lower-triangular one. The reason for this is that the m-dual basis obtained from either `LLLDualUT` or `LLLDualLT`. has a large part on its right side that is almost already lower-triangular. This is shown in more details in the `TestBasisTriSmall` example of Section 12.6. Changing the order of the calls of the two functions does not change this behavior. Finally, we also see that the old triangulation algorithm (`UppTriDualOld`) is about five times slower than the new one and that running the algorithm that constructs a triangular basis is significantly faster when the basis is already triangular (`UppTriP2` and `UppTriDual2`).

Figure 3 shows similar results for $m = 1048573$ and also some timings to compare `int64_t` vs `ZZ` for the choice of `Int`. We see that the triangularization procedures run much faster with `int64_t` than with `ZZ`, while the LLL procedures are slower.

12.6 TestBasisTriSmall

This example examines what happens when we apply LLL and construct triangular bases for the primal and m-dual lattices associated with an LCG with modulus m and multiplier a. It illustrates the properties described in Section 3.3. We want to see for instance what

[5] From Pierre: * ???

happens when $m\boldsymbol{e}_i$ becomes a shortest vector in the primal and what are the consequences on the primal and m-dual bases.

The program takes fixed values of m and a, and performs the following in $5, 10, 15, \ldots, 35$ dimensions. It first builds a primal basis `basis0` and reduces it with LLL. Then it copies it to `basis1`, builds a lower-triangular basis from `basis1` to `basis2`, computes the m-dual of this triangular basis in `basisdual`, applies LLL to this m-dual, and recovers the length of the shortest nonzero vector. Then it copies `basis0` to `basis1` again and repeats the same operations, but with upper-triangular instead of lower-triangular in `basis2`. After each operation, it prints the basis, so we can see how it looks like and how things evolve when we increase the dimension. The results for various m are in files `testBasisTriSmall*.res`.

Let us look for example at `testBasisTriSmall10-5.res`, which contains the results for $m = 1021$ with $a = 73$, with $\delta = 0.5$ for LLL. We see that in 20 dimensions already, more than half of the vectors in the reduced primal basis have the form $m\boldsymbol{e}_i$. In 25 dimensions or more, the shortest vector also has that form, so its length is $m = 1021$. The true shortest vector in the m-dual in 20 or more dimensions has length $\sqrt{3} \approx 1.732$ for this example, but LLL does not always find it. It often returns a shortest vector of length 2. In the m-dual, a simple upper-triangular basis is the identity except for the last column and a simple lower-triangular basis is the identity except for the first column. When these matrices are transformed by LLL, we get very short vectors, so there are many zeros, and the zeros lie mostly on the right side of the basis matrix. In large dimensions, the right part of the reduced basis is almost equal to the right part of the identity matrix, except for the last column when we start from the upper-triangular basis. In all cases, we see this behavior roughly beyond the first 8 to 10 columns. For larger m, such as $m = 1048573$ and $m = 1073741827$, this appears in slightly larger dimensions and after a few more columns, but the same behavior is also observable. So in large dimensions, the reduced basis is close to being lower-triangular, and this explains why transforming it to lower-triangular is faster than transforming it to upper-triangular, as we saw in Figures 2 and 3.

12.7 TestNormDecompSpeed

This example compares the two methods to obtain bounds in the BB, the triangular basis vs the Cholesky decomposition, with the L^2 and L^1 norms, for the primal and m-dual lattices associated with LCGs. The function `compareNormsDecomp` calls `testLoop` for each case, for a given m. The `testLoop` function examines `numRep` values of a. For each one, it sets the lattice object `korlat` to the associated lattice, and for each of `numSizes` selected numbers of dimensions in $\{4, 6, 8, \ldots\}$, it calls `performReduction`. The latter constructs the appropriate basis, applies BKZ to reduce it, then calls `shortestVector` to apply the BB. It also measures the time to apply BKZ+BB and the square length of the shortest vector, and add these values to counters for the given dimension. These counters are used to print results.

Figures 4 and 5 show the results for $m = 1021$ with `Real = double` and $m = 1048573$ with `Real = NTL::RR`. In Figure 4, we see that the combination of Cholesky method with

```
Compare L2 vs L1 norms, and Cholesky vs Triangular decompositions.
Types: Int = NTL::ZZ, Real = double
TestNormDecomp with m = 1021
Number of replications (different multipliers a): 1000

PRIMAL lattice,  Norm: L2NORM,  Decomposition: CHOLESKY
Num dimens:             4          6          8         10         12         14
Microseconds:        6450      13830      25889      43719      71770     112120
Aver. squares:  19172.243  68906.201 148415.328 248996.637 343256.604 464481.866
Aver. calls BB:         4          7         11         22         39         98
Total time for everything: 0.28 seconds

PRIMAL lattice,  Norm: L2NORM,  Decomposition: TRIANGULAR
Num dimens:             4          6          8         10         12         14
Microseconds:       10291      20998      38756      65731     104105     167362
Aver. squares:  19172.243  68906.201 148415.328 248996.637 343256.604 464481.866
Aver. calls BB:       172        422        828       1454       2242       3671
Total time for everything: 0.41 seconds

PRIMAL lattice,  Norm: L1NORM,  Decomposition: CHOLESKY
Num dimens:             4          6          8         10         12         14
Microseconds:        9932      47286     823962    2716689    5724406   10920108
Aver. squares:  53421.824  251337.26 703808.764 1006605.287 1024485.797 1030976.305
Aver. calls BB:         9         54        689       2220       3334       4592
Total time for everything: 20.25 seconds

PRIMAL lattice,  Norm: L1NORM,  Decomposition: TRIANGULAR
Num dimens:             4          6          8         10         12         14
Microseconds:       11493      26430      59100      89296     108892     138097
Aver. squares:  53421.824  251337.26 703808.764 1006605.287 1024485.797 1030976.305
Aver. calls BB:       284        853       2166       3143       3208       3234
Total time for everything: 0.44 seconds

DUAL lattice,  Norm: L2NORM,  Decomposition: CHOLESKY
Num dimens:             4          6          8         10         12         14
Microseconds:        6734      13448      23816      38654      59358      90120
Aver. squares:     18.683      6.805      4.963      4.162      3.859      3.743
Aver. calls BB:         4          7         14         27         62        156
Total time for everything: 0.26 seconds

DUAL lattice,  Norm: L2NORM,  Decomposition: TRIANGULAR
Num dimens:             4          6          8         10         12         14
Microseconds:       13232      27406      50007      81111     122046     184529
Aver. squares:     18.683      6.805      4.963      4.162      3.859      3.743
Aver. calls BB:       208        477       1055       1802       2825       4566
Total time for everything: 0.51 seconds

DUAL lattice,  Norm: L1NORM,  Decomposition: CHOLESKY
Num dimens:             4          6          8         10         12         14
Microseconds:       10899      42349     312071    3064502   15573982  144380934
Aver. squares:     48.752     21.984     17.722     15.485      14.31     13.988
Aver. calls BB:         9         47        383       2959      16448     134192
Total time for everything: 163.41 seconds

DUAL lattice,  Norm: L1NORM,  Decomposition: TRIANGULAR
Num dimens:             4          6          8         10         12         14
Microseconds:       14057      29198      54184      89037     131649     202759
Aver. squares:     48.752     21.984     17.722     15.485      14.31     13.988
Aver. calls BB:       256        575       1297       2169       3179       5255
Total time for everything: 0.55 seconds
```

Figure 4: Partial output of `TestNormDecompSpeed` with $m = 1021$.

```
Compare L2 vs L1 norms, and Cholesky vs Triangular decompositions.
Types: Int = NTL::ZZ, Real = NTL::RR
TestNormDecomp with m = 1048573
Number of replications (different multipliers a): 1000

PRIMAL lattice,  Norm: L2NORM,  Decomposition: CHOLESKY
Num dimens:             4          6          8         10
Microseconds:       91392     295727     729583    1518998
Aver. squares:  574748394 6882739584 2.69639e+10 6.34973e+10
Aver. calls BB:         4          6         10         18
Total time for everything: 2.64 seconds

PRIMAL lattice,  Norm: L2NORM,  Decomposition: TRIANGULAR
Num dimens:             4          6
Microseconds:    16727791   66251719
Aver. squares:  574748394 6882739584
Aver. calls BB:     24216      93236
Total time for everything: 82.98 seconds

PRIMAL lattice,  Norm: L1NORM,  Decomposition: CHOLESKY
Num dimens:             4          6          8         10
Microseconds:      108385     426770    2867497   45972817
Aver. squares: 1601226335 2.68178e+10 1.35785e+11 3.92460e+11
Aver. calls BB:         8         54        680      10777
Total time for everything: 49.38 seconds

PRIMAL lattice,  Norm: L1NORM,  Decomposition: TRIANGULAR
Num dimens:             4          6
Microseconds:    19020298   92880047
Aver. squares: 1601226335 2.681784853e+10
Aver. calls BB:     40440     189829
Total time for everything: 111.90 seconds

DUAL lattice,  Norm: L2NORM,  Decomposition: CHOLESKY
Num dimens:             4          6          8         10
Microseconds:       75651     208867     473994     956314
Aver. squares:    548.739     66.099     25.916     15.301
Aver. calls BB:         4          6         11         20
Total time for everything: 1.73 seconds

DUAL lattice,  Norm: L2NORM,  Decomposition: TRIANGULAR
Num dimens:             4          6
Microseconds:    24941961   98359657
Aver. squares:    548.739     66.099
Aver. calls BB:     29755     100178
Total time for everything: 123.30 seconds

DUAL lattice,  Norm: L1NORM,  Decomposition: CHOLESKY
Num dimens:             4          6          8         10
Microseconds:       91089     342252    2502629   32882394
Aver. squares:   1523.097    256.456    126.819      85.13
Aver. calls BB:         8         56        683       8759
Total time for everything: 35.83 seconds

DUAL lattice,  Norm: L1NORM,  Decomposition: TRIANGULAR
Num dimens:             4          6
Microseconds:    26962131  113116951
Aver. squares:   1523.097    257.224
Aver. calls BB:     42872     146147
Total time for everything: 140.08 seconds
```

Figure 5: Partial output of `TestNormDecompSpeed` with $m = 1048573$.

the L^1 norm takes much more time than the other ones, for both the primal and m-dual, especially in 8 or more dimensions. For this example, the triangular method takes about the same time for the L^1 and L^2 norm. the Cholesky method is faster for the L^2 norm but slower for the L^1 norm.

Figure 5 shows how things change with the triangular method when we increase m. The timings for that method increase tremendously. This is the first reason why we ran this method only for 4 and 6 dimensions. The second reason, which is also the reason why we used `Real = NTL::RR` for that case, is that with this method, the bounds on the z_j's can take extremely large values when m and the dimension are large. We have seen this in Example 3 already and we see it again here, as follows. [6]

Example 5. If we use `Real = double` with $m = 1048573$, for $a = 426593$, in $t = 6$ dimensions, for the m-dual lattice with the triangular method, with either norm, the `TestNormDecompSpeed` program appears to run forever. Why is that? The lower-triangular m-dual basis for that case is

$$\boldsymbol{L} = \begin{pmatrix} 1048573 & 0 & 0 & 0 & 0 & 0 \\ -426593 & 1 & 0 & 0 & 0 & 0 \\ -180317 & -313423 & 1 & 0 & 0 & 0 \\ 462641 & -48144 & -290030 & 1 & 0 & 0 \\ 419429 & 69904 & 269633 & 459375 & 1 & 0 \\ 524285 & 524282 & -1 & -524286 & -524281 & 1 \end{pmatrix}.$$

After BKZ, the L^2 norm of the shortest vector in that lattice is $b(2) = \sqrt{51}$. The bounds on z_6 are then $0 \le z_6 \le \lfloor \sqrt{51}/\ell_{6,6} \rfloor = 7$. By putting `red.setVerbosity(4)` for the reducer in the code, we can see a trace of all the visited nodes in the BB tree and the bounds computed at each node. Figure 6 shows a small piece of that large trace. The index j in that figure is 1 less than in this guide, because the indices in the code start at 0 instead of 1. We see that at some stage in the BB algorithm, we have $z_6 = 0$, $z_5 = 1$, $z_4 = -459375$, and the bounds for z_3 are $-133232800890 \le z_3 \le -133232800876$. When we take the middle value $z_3 = -133232800883$, we get the bounds $-41758246267372424 \le z_2 \le -41758246267372409$. These values are about 4.1758×10^{16} and are not represented exactly in a `double`. For this reason, they are in fact incorrect. The correct values (obtained with `Real = NTL::RR`) are $-41758246267372420 \le z_2 \le -41758246267372406$. At that point, the program will continue executing with the wrong bounds and probably make similar errors in many other places.

Many steps later, we have at some point $z_6 = 1$, $z_5 = 524281$, $z_4 = -240841060089$, we get the bounds $-69851274021071552 \le z_3 \le -69851274021071537$, and with $z_3 = -69851274021071544$, we get $-9223372036854775808 \le z_2 \le 9223372036854775807$, which is not only incorrect, but gives an enormous interval. Basically, the program will run forever

[6] From Pierre: *** We should make more extensive testing to check if this oscillatory behavior can also occur with the Cholesky decomposition and when. It probably does if the pre-reduction is not good enough.

```
   ⋮
tryZ: vector z: [0, 0, 0, -459375, 1, 0]
  j=2,  center[j]= -1.33233e+11, min= -133232800890, max= -133232800876
tryZ: vector z: [0, 0, -133232800883, -459375, 1, 0]
  j=1,  center[j]= -4.17582e+16, min= -41758246267372424, max= -41758246267372409
tryZ: vector z: [0, -41758246267372416, -133232800883, -459375, 1, 0]
  j=0,  center[j]= -1.69886e+16, min= -16988611736010400, max= -16988611736010401
   ⋮
tryZ: vector z: [0, 0, 0, 0, 0, 1]
  j=4,  center[j]= 524281, min = 524274, max= 524288
tryZ: vector z: [0, 0, 0, 0, 524281, 1]
  j=3,  center[j]= -2.40841e+11, min= -240841060096,  max= -240841060082
tryZ: vector z: [0, 0, 0, -240841060089, 524281, 1]
  j=2,  center[j]= -6.98513e+16, min= -69851274021071552, max= -69851274021071537
tryZ: vector z: [0, 0, -69851274021071544, -240841060089, 524281, 1]
  j=1,  center[j]= -2.1893e+22, min= -9223372036854775808, max= 9223372036854775807
```

Figure 6: Pieces of a detailed trace from `TestNormDecompSpeed` with $m = 1048573$.

to try all those values of z_2. This occurs because the true bounds in absolute value are too large to be represented correctly. It can be corrected by using the `NTL::RR` representation, but it will be slow. This is an important drawback of computing the BB bounds with the triangular method. For large m, using the Cholesky decomposition is usually much faster.

12.8 TestReducersSpeed

This example compares the performances of various pre-reduction strategies when finding a shortest nonzero vector with the BB combined with the Cholesky method, for both the primal and the m-dual of a Korobov `Rank1Lattice`. In the code, we use template parameters for the types, so we can compare different `Real` types in the same program execution. The top function called by the `main` is `comparePreRed`. It calls `testLoop` for the primal lattice, then for the m-dual. The function `testLoop` makes the experiment for a given m, a given norm type and a given type of decomposition, for either the primal or the m-dual. It examines `numRep` values of the multiplier a of the corresponding LCG. For each choice of a and each selected number of dimensions, it calls `compareManyReductions` to try and compare various reduction methods. For each method, the `performReduction` function first (re)builds the appropriate basis (primal or m-dual) from scratch for the relevant number of dimensions. In the m-dual case, it calls `dualize` to put this m-dual basis in place of the primal basis. Then it applies the selected reduction methods and recovers in `len2` the square length of the shortest basis vector. In case the BB procedure is applied, it also recovers the number of calls to the recursive BB procedure (the number of visited nodes in the BB tree) and the number of those nodes that are tree leaves (at which we test a fully specified vector $\boldsymbol{z}$). These values as well as the CPU time, are added to counters that cumulate the sum of CPU times, the sum of square lengths, the number of visited nodes, and the number of visited

leaves, for each dimension and each method. The sums of CPU times and the average per run for the other counters are printed in tables after the experiment is completed.

The function `compareManyReductions` tests 18 different reduction methods. Each one has a number `meth` and a short name `names[meth]` given in an array at the beginning of the program. The first four methods apply only LLL or BKZ reductions and do not run the BB algorithm. For this reason, they are much faster, but they do not always find a shortest nonzero lattice vector. All other methods, whose short names end by "+BB", run the BB algorithm after the pre-reduction and always find a shortest vector. Note that the BKZ function always starts internally by applying LLL with the same δ, so there is no need to do it explicitly. Recall that the results of Figure 1 suggested that an *incremental strategy* that first makes one or more LLL pass(es) with a smaller δ could give better results than applying LLL directly with the target δ. This was with LLL only, no BKZ was applied, and it is unclear a priori if this incremental strategy can be worthwhile when we do BKZ and BB. To see that, the fourth method applies an incremental strategy without the BB, and the last three methods do that before the BB. L5+L9+BKZ-k means LLL with $\delta = 0.5$ followed by LLL with $\delta = 0.9$ followed by BKZ with $\delta = 0.99999$ and blocksize k. L8+BKZ-k is similar, but does LLL only with $\delta = 0.8$.

Table 5: Total times in seconds for the `TestReducersSpeed` example. The timings for up to 70 dimensions for $m = 1021$ and up to 40 dimensions for the other values of m.

Code	`<Int, Real>`	$m = 1021$		$m = 1048573$		$m = 1099511627791$	
		primal	m-dual	primal	m-dual	primal	m-dual
LD	`<long, double>`	7.3	169.8	—	—	—	—
ZD	`<ZZ, double>`	10.6	181.5	10.9	76.9	183.0	250.9
ZQ	`<ZZ, quad_float>`	61.1	1715.1	80.3	636.7	1562.2	2155.4
ZR	`<ZZ, RR>`	—	—	615.0	4682.7	—	—

Numerical results are in files `examples/results/testRedSpeed*.res`, where "`*`" is 10 for $m = 1021$, 20 for $m = 1048573$, and 40 for $m = 1099511627791$. We took `numRep` $= 100$ different multipliers a for $m = 1021$ and `numRep` $= 50$ for the other m.

Figures 7, 8, and 9 show partial results for the m-dual, for $m = 1099511627791$, 1048573, and 1021, respectively, with the `<ZZ, double>` types. Figure 10 gives partial results for the primal lattice for $m = 1021$ with the `<int64_t, double>` types. For each column of each table, the fastest time among the +BB methods and the smallest number of visited nodes are indicated by a red star.

The following comments apply generally to all cases that we tried, and not only to the cases shown in the figures. The reader is encouraged to look at the detailed results in the `.res` files and perhaps try other cases to explore. In less than 20 dimensions, there is not much difference in speed between the different methods, and the BB does not take much extra time, because in most cases we already have a shortest vector before the BB and relatively few nodes are visited in the BB tree.

```
TestReducersSpeed with m = 1099511627791
Types: Int = NTL::ZZ, Real = double
Number of replications (different multipliers a): 50
DUAL lattice,  Norm: L2NORM,  Decomposition: CHOLESKY.

Num. dimensions:       5        10        20        30        40

Computing times in microseconds:
LLL5                  667      2119      5420      7349     11160
LLL99999              586      2532     13435     32540     46944
BKZ99999-10           575      2509     17465     67577    168436
L5+L9+BKZ-10          663      2768     17659     65919    162216
LLL99999+BB           660      3207     19430    208532  73554207
BKZ99999-6+BB         599      3177     22393*   144172* 19888233
BKZ99999-8+BB         551      3031     23161    149543  15309212
BKZ99999-10+BB        528      2926     23701    154806  16261578
BKZ99999-12+BB        517      2871     24310    152247  14199518
BKZ999-6+BB           520      2821     22428    148173  21815602
BKZ999-8+BB           503      2787     23103    152991  16436182
BKZ999-10+BB          503      2767     23586    156544  17473434
BKZ999-12+BB          494*     2731*    24385    150444  13184568
L8+BKZ-10+BB          651      3318     22698    147323  13987731
L5+L9+BKZ-10+BB       742      3496     23571    157068  13822829
L5+L9+BKZ-12+BB       674      3355     24001    160298  12090483*

Average square length of shortest basis vector:
LLL5             36783.70     255.0     46.82     40.64     36.38
LLL99999         36395.12     231.9     25.38     14.84     13.50
BKZ99999-10      36395.12     231.4     25.16     14.28     11.94
L5+L9+BKZ-10     36395.12     231.4     25.14     14.26     11.86
All +BB methods  36395.12     231.4     25.14     14.10     11.50

Average number of calls to the recursive BB procedure 'tryZ':
LLL99999+BB             5        16       793    129459  58161623
BKZ99999-10+BB          5        16       701     57957  12646861
BKZ99999-12+BB          5        16       693     53608  11006050
BKZ999-10+BB            5        16       697     58726  13659907
BKZ999-12+BB            5        16       680     52185* 10183027
L8+BKZ-10+BB            5        16       687     56562  10830150
L5+L9+BKZ-12+BB         5        16       668*    56555   9309786*

Average number of visited leaves in the BB procedure:
LLL99999+BB             1         1         1         2         7
BKZ99999-12+BB          1         1         1         2         5
BKZ999-12+BB            1         1         1         2         5
L5+L9+BKZ-12+BB         1         1         1         2         5

Total time for everything: 250.9 seconds
```

Figure 7: Partial output of TestReducersSpeed for $m = 1099511627791$.

```
TestReducersSpeed with m = 1048573
Types: Int = NTL::ZZ, Real = double
Number of replications (different multipliers a): 50
DUAL lattice,  Norm: L2NORM,  Decomposition: CHOLESKY.

Num. dimensions:       5        10        20        30        40

Computing times in microseconds:
LLL5                  458      1361      2762      4186      6016
LLL99999              398      1634      7229     12474     15153
BKZ99999-10           394      1645     12111     43211     70827
L5+L9+BKZ-10          512      2122     13068     43997     78907

LLL5+BB               470      1981     14013    592737  14933865
LLL8+BB               456      2163     11524*   322847  14179214
LLL99999+BB           411      2196     13004    196796   8322403
BKZ99999-6+BB         415      2219     15892    134600   4236738
BKZ99999-8+BB         388      2055     16860    127164*  3801107
BKZ99999-10+BB        381      1946     17702    142975   3044588
BKZ99999-12+BB        383      1892     18164    139357   2517072*
BKZ999-6+BB           378      1846     16026    141825   4242654
BKZ999-8+BB           370      1823     16866    133130   3398739
BKZ999-10+BB          369*     1792     17537    137647   2883433
BKZ999-12+BB          369*     1775*    17957    143816   3116050
L8+BKZ-10+BB          477      2409     17808    140802   3296159
L5+L9+BKZ-10+BB       550      2756     18568    137954   2998657
L5+L9+BKZ-12+BB       500      2583     18942    139505   3138813

Average square length of shortest basis vector:
LLL5               152.02    16.12      8.68      7.78      7.04
LLL99999           151.14    15.46      6.98      5.96      5.62
BKZ99999-10        151.14    15.46      6.84      5.78      5.34
L5+L9+BKZ-10       151.14    15.46      6.82      5.78      5.36
All +BB methods    151.14    15.46      6.82      5.74      5.24

Average number of calls to the recursive BB procedure 'tryZ':
LLL5+BB                5        27      5303    460653  11639706
LLL8+BB                5        22      1853    243822  11153257
LLL99999+BB            5        21      1321    135953   6530546
BKZ99999-10+BB         5        21      1081     68703   2296208
BKZ99999-12+BB         5        21      1051     60097*  1861705*
BKZ999-10+BB           5        21      1057     64542   2161005
BKZ999-12+BB           5        21      1045     62695   2335329
L5+L9+BKZ-12+BB        5        21      1028*    60235   2349325

Average number of visited leaves in the BB procedure:
LLL5+BB                1         1         6        13        31
LLL8+BB                1         1         2         7        21
LLL99999+BB            1         1         2        10        26
BKZ99999-10+BB         1         1         2         6        27
BKZ99999-12+BB         1         1         2         7        28
L5+L9+BKZ-12+BB        1         1         2         6        22

Total time for everything: 77.4 seconds
```

Figure 8: Partial output of TestReducersSpeed for $m = 1048573$ and m-dual lattice.

```
TestReducersSpeed with m = 1021
Types: Int = NTL::ZZ, Real = double
Number of replications (different multipliers a): 100
DUAL lattice,  Norm: L2NORM,  Decomposition: CHOLESKY.

Num. dimensions:  10        20        30        40        50        60        70
Computing times in microseconds:
LLL5              1827      3665      6058      9225     13556     19302     26471
LLL99999          2290      7456     11070     15114     20430     27507     36507
BKZ99999-10       2617     17470     42294     74845    105366    143086    193989
L5+L9+BKZ-10      3520     19966     49724     84590    117407    163059    222943
LLL5+BB           3078     16408*    82218*   266307    975886   3601698  10808413
LLL8+BB           3227     17263     92379    309017   1296846   5143418  18017647
LLL99999+BB       3286     18869     90633    303092   1328958   6102461  20877337
BKZ99999-6+BB     3513     24665     96293    265603    799623   2822246   8168659
BKZ99999-8+BB     3200     26173     93132    247950    713821   2659301   7862103
BKZ99999-10+BB    3048     27396     97009    206098    573580   2061635   6030053
BKZ99999-12+BB    2902     29975     99822    218098    509552   1662647*  4726783*
BKZ999-6+BB       2793     24435     96462    256870    808835   2886718   8696810
BKZ999-8+BB       2748     26077     92615    254123    725117   2697742   7726230
BKZ999-10+BB      2708     27158     97193    204866*   600652   2087770   5954333
BKZ999-12+BB      2665*    29843     99448    223191    497162*  1708653   5095630
L8+BKZ-10+BB      3939     30015    100608    213996    644618   2157627   6816609
L5+L9+BKZ-10+BB   4583     31333    106291    230656    651904   2412697   8031166
L5+L9+BKZ-12+BB   4219     32405    111543    241286    533280   1686852   5074948

Average square length of shortest basis vector:
LLL5              4.34      3.37      3.17      3.06      2.96      2.89      2.83
LLL99999          4.23      3.19      2.98      2.87      2.81      2.76      2.76
BKZ99999-10       4.23      3.17      2.90      2.79      2.76      2.76      2.76
All +BB methods   4.23      3.17      2.90      2.78      2.76      2.76      2.76

Average number of calls to the recursive BB procedure `tryZ`:
LLL5+BB             36      2001     21887     88561    360852   1359010   4011824
LLL8+BB             31      1691     24416    102026    479114   1901877   6543645
LLL99999+BB         29      1446     22133     97435    482462   2242583   7374763
BKZ99999-12+BB      29      1130     11443     37710*   129822    518054   1481856*
BKZ999-12+BB        29      1125     11198*    38739    125417*   536394   1612499
L5+L9+BKZ-12+BB     29      1097     12220     38914    127645    506872*  1566186

Average number of visited leaves in the BB procedure:
LLL5+BB              1         7        18        20        22        39        63
LLL99999+BB          1         6        17        22        24        42        70
BKZ99999-12+BB       1         6        19        19        25        49        77

Total time for everything: 181.5 seconds
```

Figure 9: Partial output of TestReducersSpeed for $m = 1021$ and m-dual lattice.

```
TestReducersSpeed with m = 1021
Types: Int = int64_t, Real = double
Number of replications (different multipliers a): 100
PRIMAL lattice,  Norm: L2NORM,  Decomposition: CHOLESKY.

Num. dimensions:    10        20        30        40        50        60        70
Computing times in microseconds:
LLL5              1060      2847      5128      8127     12069     18335     25130
LLL99999          1489      6902     12563     20421     29585     44302     56788
BKZ99999-10       1764     15820     32640     52159     76208    109341    152796
LLL5+BB            927     10286     21196     25657     30972     39236     48592
LLL99999+BB       1353     10407     21820     30683     40706     58698     74091
BKZ99999-10+BB    1408     18219     41236     63503     88447    124551    171087
BKZ999-10+BB      1324     18204     42305     63071     89419    123231    171683
L5+L9+BKZ-12+BB   1594     20496     46809     67890     96173    138186    192576

Average square length of shortest basis vector:
LLL5          266368.4 912875.4 1004531.57 1009269.11 1014006.65 1018744.19 1023481.73
All +BB       253621.6 860224.5 1004531.57 1009269.11 1014006.65 1018744.19 1023481.73

Average number of calls to the recursive BB procedure 'tryZ':
LLL5+BB             29      3004      6528      6828      6972      6831      6617
LLL99999+BB         23      1364      3533      3450      3305      3642      3578
BKZ99999-10+BB      23      1200      3251      3759      3555      3863      4034
L5+L9+BKZ-12+BB     23      1146      3205      3623      3577      3880      4063

Average number of visited leaves in the BB procedure:
LLL5+BB              1         2         4         4         4         4         4
LLL8+BB              1         2         6         6         6         6         6
L5+L9+BKZ-12+BB      1         1         6         6         6         6         6

Total time for everything: 7.3 seconds
```

Figure 10: Partial output of TestReducersSpeed for $m = 1021$ in the primal lattice, with `Int = int64_t`.

Applying only LLL before the BB is often the fastest approach in up to 20 dimensions, and also in more dimensions for $m = 1021$, but for the larger values of m in larger dimensions, it is much slower, even with $\delta = 0.99999$. In that case, doing a BKZ reduction before the BB is more effective, and using a larger blocksize k (around 10 or 12) seems more important than taking δ closer to 1. Compare BKZ99999 with BKZ999 in Figures 7 and 8, for example. A block size larger than 12 may be even better in larger dimension; see the `TestCholesky` example for an illustration. For the larger m in the m-dual, we removed LLL5+BB and LLL8+BB because with these methods, the program was taking way too much time to run (the BB was much too slow) and sometimes failed. The incremental strategy does not provide much improvement when followed by BKZ and BB. It wins only for the largest m in 40 dimensions, and not by a large margin. Applying only BKZ with a large enough k for the pre-reduction performs quite well in general. The case of $m = 1021$ for the primal lattice (Figure 10) behaves differently than others: in up to 70 dimensions, the timings and the number of visited nodes do not increase much with the dimension, and also applying just LLL5 as a pre-reduction is the fastest approach. This is interesting because in general, in the worst case, the BB should take exponential time as a function of the dimension.

For the average square lengths, all the "+BB" methods give the same values because they always find a shortest vector, so we show only one row for these. The values are also the same across the different types codes. When comparing these (exact) values for the m-dual lattices, we see that without the "+BB" we do not always find a shortest vector; we get larger values even with $\delta = 0.99999$. In Figure 7, for example, in 30 and 40 dimensions, the average square length is more than three times larger after LLL5 only than after BB. This is very significant!

For $m = 1099511627791$, if we look at the result files and compare, we also find that the number of visited BB nodes is slightly different for the different `Real` representations. This is due to the (small) numerical imprecision when we perform the Cholesky decomposition and we compute the bounds in the BB algorithm, as explained in Section 8.4. For these experiments, we used $\epsilon = 0$. We also tried with $\epsilon = 10^{-6}$ instead, and the number of visited BB nodes was the same in almost all cases (in a few cases it changed by 1 or 2), and the shortest vector lengths remained the same in all cases.

Table 5 gives the total time to run the experiment for each selection of types. The running times are slower for `Real = quad_float` compared with `double`, and much slower with `RR`, due to the extra precision. We also see in Table 5 that the total running times are shorter for the primal lattice than for the m-dual, and this is especially true for smaller m. On the other hand, the shortest vectors are much longer in the primal than in the dual. On closer examination, we find that the BB takes more time in the m-dual, but the LLL and BKZ reductions alone are faster in the m-dual than in the primal. Overall, we found in these experiments that when BKZ is applied before the BB, doing an incremental LLL reduction before the BKZ reduction does not help much. This differs from what we saw in Figure 1, where no BKZ was done.

In addition to these 18 methods, we also tried BB with no pre-reduction at all (Direct BB) and with pairwise pre-reductions only (Pairwise+BB) but with these methods the BB

was much too slow and often failed because the BB tree had too many nodes (more than one billion), in 30 and 40 dimensions. With the larger modulus, in 40 dimensions, the BB also fails most of the time when done after LLL5 and occasionally when done after LLL8, also because the number of visited BB nodes is excessive. This is why these two methods do not appear in the table for the larger m. So we really need to perform serious reduction work before applying the BB. With $m = 1099511627791$ and `Real = double`, the BB after no pre-reduction also fails most of the time even in 5 dimensions because the Cholesky decomposition gives negative elements on the diagonal. With a large m, we need more precision for the real numbers, but a good pre-reduction is very important, as we shall see again in the next example.

12.9 TestCholesky

This example tests the LDL Cholesky decomposition of the matrix of scalar products with different floating-point representations. The program construct a basis for the m-dual lattice associated with a LCG, applies a pre-reduction to that basis, then performs the LDL Cholesky decomposition, and prints some information, including the component $\boldsymbol{D}$ of the decomposition and the bound $\delta_t(2)$ on z_t at the first level of the BB. If the `doBB` variable is set to `true` in the `main`, it also runs the BB algorithm, counts the number of visited nodes in the BB tree, and prints some of the bounds on the z_j's computed in the BB algorithm.

We ran the program for the 40-bit prime $m = 1099511627791$, $a = 401173573$, in 5, 30, and 40 dimensions, with the following three choices of pre-reductions: no pre-reduction, `LLL5`, `LLL999`, `BKZ99999-12`, and `BKZ99999-20`, defined as in the previous example. The selected types were `Int = ZZ` with `Real = double, quad_float, RR`. By default, `RR` has 150 bits of precision, but in 40 dimensions we also tried `RR` with 250 bits of precision, to compare. The complete output is in file `testCholesky40`.

Table 6 shows some values taken from the output file. It gives the values of d_t and $\delta_t(2)$ defined in Section 8.3, and the number of visited nodes in the BB tree. With "no pre-reduction", if we take `Real = double`, the Cholesky factorization fails for all numbers of dimensions (we get negative diagonal elements due to numerical errors), so no number is given for that case. For the other `Real` types and no pre-reduction, the Cholesky factorization works and the values differ only slightly between the two types, but the bounds $\delta_t(2)$ on z_t are extremely large. Already at the first level of the BB tree, more than 245 million values of z_t must be examined when $t = 5$ (first row of the table), and more than one billion when $t \geq 30$ (fourth row). The full BB algorithm will then take forever to run. This algorithm stops immediately when one billion nodes have been examined.

With an LLL or BKZ pre-reduction, the bounds are much smaller, and the values of d_t and $\delta_t(2)$ are the same (up to the given numbers of digits) for all real number representations. That is, the LDL Cholesky method is pretty stable after the basis has been reduced sufficiently, at least for this example. For $t = 5$, they are also the same for all types of pre-reduction, because all pre-reductions give the same basis in that case. For this reason, we display only two cases in the table. For the larger values of t, however, the BKZ methods

Table 6: Some values of d_t and $\delta_t(2)$ obtained for Example `TestCholesky`.

t	pre-reduction	Real	d_t	$\delta_t(2)$	nodes in BB
5	none	`RR`	2.673953322	245332681.4	$> 10^9$
5	`LLL5`	`double`	139169.2263	0.449210753	5
5	`BKZ99999-12`	`RR`	139169.2263	0.449210753	5
30	none	`RR`	0.1578115561	1009864058	$> 10^9$
30	`LLL5`	`double`	0.1990594272	15.6894119	5628538
30	`LLL5`	`quad_float`	0.1990594272	15.6894119	5628539
30	`LLL5`	`RR`	0.1990594272	15.6894119	5628538
30	`LLL999`	`double`	0.9004257043	3.495202968	24945
30	`LLL999`	`quad_float`	0.9004257043	3.495202968	24945
30	`LLL999`	`RR`	0.9004257043	3.495202968	24945
30	`BKZ99999-12`	`double`	2.383026895	2.148482748	18466
30	`BKZ99999-12`	`quad_float`	2.383026895	2.148482748	18467
30	`BKZ99999-12`	`RR`	2.383026895	2.148482748	18466
30	`BKZ9999999-20`	`double`	2.227871426	2.222036972	10104
30	`BKZ9999999-20`	`quad_float`	2.227871426	2.222036972	10104
30	`BKZ9999999-20`	`RR`	2.227871426	2.222036972	10104
40	`LLL5`	`double`	0.1415697397	18.60427812	$> 10^9$
40	`LLL5`	`RR`	0.1415697397	18.60427812	$> 10^9$
40	`LLL999`	`double`	0.2163843044	7.129900427	20920057
40	`LLL999`	`quad_float`	0.2163843044	7.129900427	20920055
40	`LLL999`	`RR`	0.2163843044	7.129900427	20920059
40	`LLL999`	`RR-250`	0.2163843044	7.129900427	20920054
40	`BKZ99999-12`	`double`	1.107460778	3.151607791	18583011
40	`BKZ99999-12`	`quad_float`	1.107460778	3.151607791	18583012
40	`BKZ99999-12`	`RR`	1.107460778	3.151607791	18583010
40	`BKZ99999-12`	`RR-250`	1.107460778	3.151607791	18583013
40	`BKZ9999999-20`	`double`	0.815408147	3.672897591	5518423
40	`BKZ9999999-20`	`quad_float`	0.815408147	3.672897591	5518422
40	`BKZ9999999-20`	`RR`	0.815408147	3.672897591	5518422
40	`BKZ9999999-20`	`RR-250`	0.815408147	3.672897591	5518423

produce narrower bounds than LLL, and are better in terms of the number of visited nodes. We also see that `BKZ99999-20` does better than `BKZ99999-12` in large dimensions. When a pre-reduction is applied, the different types of `Real` representations give almost all the same number of visited nodes in the BB, except for small variations. The small variations are due to small numerical imprecision that depend on the `Real` type. To reduce the chances of missing valid tree nodes due to small numerical errors, we can set the safety margin `epsBounds` in the program to a value slightly larger than zero, for example 10^{-6}. This notion is discussed in Section 8.4. We re-ran the entire experiment with `epsBounds`$= 10^{-6}$ and the shortest vector lengths did not change. Moreover, the number of visited nodes were exactly the same as in Table 6 for all cases!

12.10 TestFOMPairs

In this example, we explore the behavior of the FOM (31) for LCGs with modulus m and multiplier a, for several values of a. We take the subsets defined in (35), so only the projections that contain the first coordinate are considered, and we consider two choices for the vector $\boldsymbol{t} = (t_1, \ldots, t_d)$. For each a, we compute the FOM for the primal and for the m-dual and we find the projection for which the minimum is attained (the worst projection) in each case.

The program examines `numRepVerb` $= 1000$ multipliers a and prints some results in a data file, one line for each a. In the first four columns, it gives the FOM and the dimension of the worst-case projection for both the primal and m-dual. In the fifth column, if the worst-case projection was the same for both the primal and m-dual, it prints the dimension of that projection, otherwise it prints 0. This data file is then used by LaTeX/PGFplots to make scatter plots of the primal vs m-dual FOMs, using colors that depend on the value in the fifth column.

The program also counts how many times the worst-case projection has s dimensions, for each s, for both the primal and the m-dual, and gives this info on the terminal. It also displays the FOMs and the worst-case projections for the first `numRepVerb` $= 10$ multipliers.

We take $m = 1048573$. The two choices of $\boldsymbol{t}$ that are considered are $\boldsymbol{t}_{16} = (16, 16, 12, 10)$ with $d = 4$ and $\boldsymbol{t}_{32} = (32, 32, 16, 12, 10)$ with $d = 5$. The number of projections of each type (the cardinalities of $S_s^{(1)}(t_s)$ for $s = 1, \ldots, d$) is (12, 15, 55, 84) in the first case and (27, 31, 120, 220, 210) in the second case (see Section 10). Table 7 gives the proportion of the projections that are in s dimensions, multiplied by 1000, and the number of times that the worst-case projection had s dimension, for each case. We see that the worst-case was for a two-dimensional projection more often than what we would expect if it was proportional to the proportion (indicated in the row "proportion $\times 1000$"), and less often for the projections in more then three dimensions. The high-dimensional projections over successive coordinates were very rarely the critical ones (it occurred only in one case). This behavior suggests that when making a search with the possibility of early exit when a bad projection is found, it should be much more effective to start with the low-dimensional projections, because they are more likely to eliminate candidates and also the values are much faster to compute for

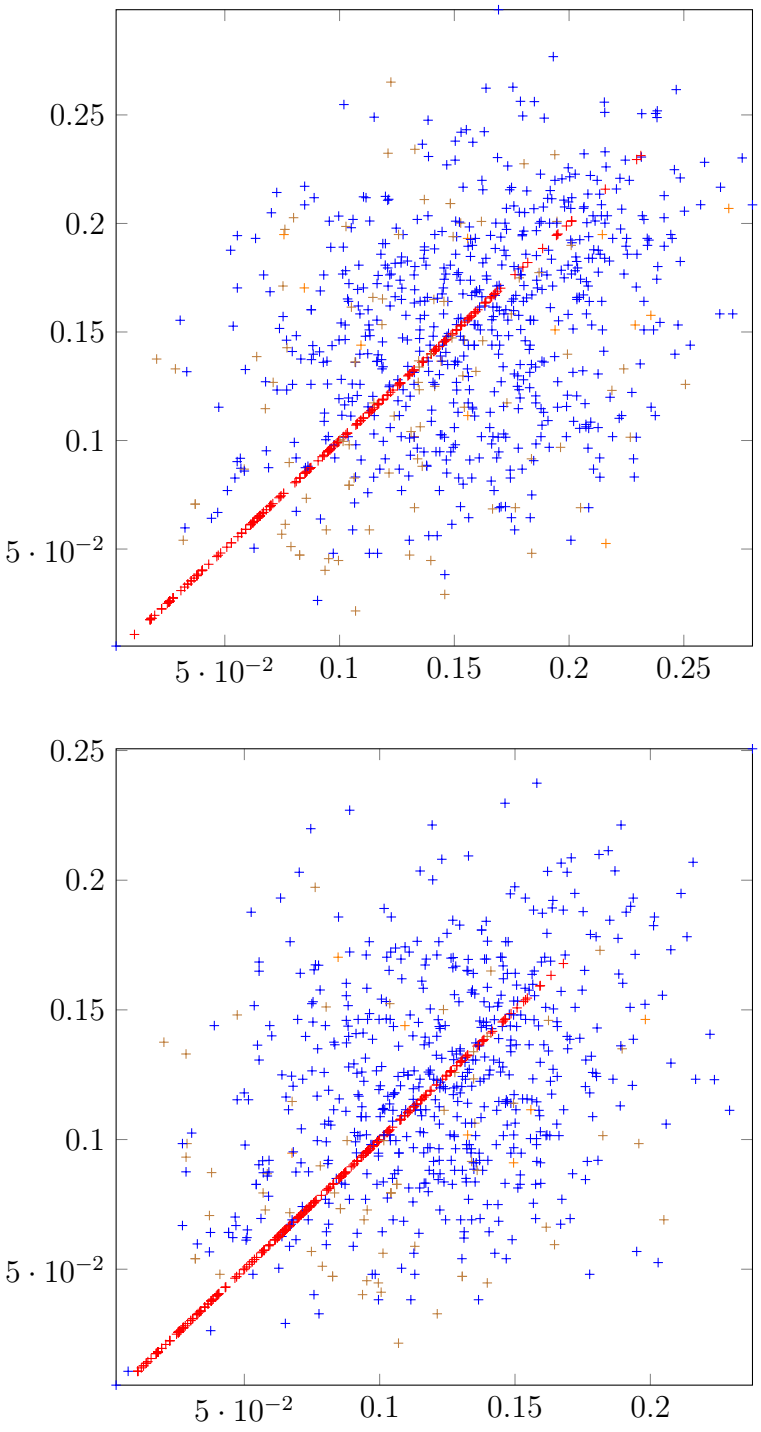

Figure 11: Scatter plots for the `TestFOMPairs` example for $m = 1048573$, with $\boldsymbol{t}_{16}$ (above) and $\boldsymbol{t}_{32}$ (below)

Table 7: Distribution of the dimension of worst-case projection for the primal and the m-dual, in Example `TestFOMPairs`.

		dimension					
		2	3	4	5	6	> 6
$\boldsymbol{t}_{16}$	proportion $\times 1000$	90.3	331.3	506.0	6.02	6.02	60.3
	num. primal	359	433	207	1	0	0
	num. m-dual	370	469	161	0	0	0
$\boldsymbol{t}_{32}$	proportion $\times 1000$	51.0	197.4	361.8	345.4	1.64	42.8
	num. primal	427	392	157	24	0	0
	num. m-dual	468	417	105	10	0	0

those projections. Of course, the behavior might differ for other types of constructions than LCGs and for other choices of $\boldsymbol{t}$.

Figure 11 shows the two scatter plots. Each one has 1000 points (x, y) where x is the FOM in primal and y is the FOM in m-dual. The blue marks are for when the worst projection is not the same for the primal and m dual. The brown and orange marks are for the cases where the worst projection is the same and is in three and four dimensions, respectively. The red marks are for when the worst projection is the same and is in two dimensions. All these red marks are on the diagonal, which means that the primal and m-dual FOMs are always exactly the same! To understand why, it suffices to examine the basis vectors for these two-dimensional projections by looking at the matrices in (2) and (3). For the projection over coordinates $\{1, s\}$, the basis vectors are $(1, a_s)$ and $(0, m)$ where $a_s = a^{s-1} \bmod m$, and the m-dual basis vectors are $(-a_s, 1)$ and $(m, 0)$. These primal and m-dual lattices are essentially the same lattice, mirrored about the horizontal axis and with the two coordinates exchanged. This means that for each two-dimensional projection, the lattice structure in the primal and m-dual is exactly the same and the length of the shortest nonzero vector is always the same.

Apart from the red points which give a perfect correlation between the primal and m-dual FOMs, the other points only show a weak correlation. We see that the best values of a for the primal FOM and for the dual FOM are not the same. There are a few points in the lower-left corner (both FOMs are bad) and in the upper-right corner (both FOMs are good), and no points in the two other corners, which is reassuring, because it means that there are no case where the FOM is excellent in the m-dual and very bad in the primal, or vice-versa.

12.11 TestFOMSearch

This example compares different ways of making a search for a good LCG multiplier a for a given prime modulus m in terms of the FOM $M_{\boldsymbol{t}}$ for a given vector $\boldsymbol{t} = (t_1, \ldots, t_d)$, with coordinate 1 included in all the projections, for either the primal or the m-dual lattices.

The FOM can be computed exactly using the BKZ+BB, or only approximated by using either LLL of BKZ only. When computing the FOM for a given a, we can either exit the procedure (early discard) as soon as we know that the FOM will be too small, or we can always complete the computations. The multipliers a that are considered can be read from a file (from a previous selection), or they can be generated as the successive powers of a given integer a_0, modulo m. The function `findBestFOMs` supports all these possibilities. It examines `numMult` values of a and makes a list of the `numBest` best ones based on the FOM `fom`. The FOM can be either for the primal or for the m-dual, depending on how it was created. It can be for the L^2 or L^1 norm, depending on which norm is used in the lattice objects for which we compute the FOM. [7]

The function `compareSearchMethods` tests and compares different ways of making the search, for a given FOM, for either the primal or the m-dual lattices. *Method 1* is a naive approach used as a basis for comparison. It computes all the terms of the FOM for all values of a (no early discarding) using BKZ+BB with $\delta = 0.99999$ and $k = 10$ in `FigureOfMerit` to get the exact FOM value in each case. This takes a lot of time. To avoid excessive computing times, for this method we take `numMult` 100 times smaller and we multiply the timings by 100 estimate the required time with the larger `numMult` value. *Method 2* also applies BKZ+BB with the the same parameters, but it uses early discarding. *Method 3* applies LLL+BB with $\delta = 0.99999$ and also use early discarding. *Method 4* uses two stages with early discarding at each stage. In the first stage, it uses only LLL with $\delta = 0.99999$ and it retains the `numBest0` multipliers in a list. In the second stage, it tests these retained multipliers from the list using BKZ+BB and it returns the `numBest` best ones in the final list. *Method 5* does the same as Method 4, except that in the first stage, it looks only at a restricted number of projections by using a vector $\boldsymbol{t}_0$ of smaller values instead of $\boldsymbol{t} = (t_1, ..., t_d)$. One of the goals is to see how much time can be saved by using early discarding and/or a two-stage method with an intermediate list.

The program runs this experiment for the primal lattices, then for the m-duals. We tried it with $m = 1048573$ (a prime number near 2^{20}) and then with $m = 1099511627791$ (a prime number near 2^{40}), in both cases with $a_0 = 91$ (a primitive element mod $m = 1048573$), $\boldsymbol{t} = (32, 32, 16, 12, 10)$, $\boldsymbol{t}_0 = (4, 32, 16, 12)$, `numMult` $= 100,000$, `numBest` $= 3$, `numBest0` $= 50$.

Table 8 summarizes the results. It gives the CPU time (in seconds) for each method, for the primal and the m-dual, for each m. For the first (naive) method, the actual times have been multiplied by 100 to estimate the required time for `numMult` $= 100,000$, while the best FOM is for the reduced number of trials, which was only 1000. For Methods 4 and 5, we give the timing for each stage. We see that the second stage takes negligible time compared to the first stage. The results are very similar for the two values of m.

Method 1 is clearly not competitive. Methods 4 and 5 take significantly more time than Methods 2 and 3, which may be surprising because intuition may suggest that doing a first

[7] From Pierre: We should also test the L^1 norm in this example and add a small table that complements Table 7 by show results and timings for the L^1 norm, on the same computer as other results. Showing results for method 2 (BKZ + BB, with discrad) would be sufficient. Of course, we cannot go to high dimensions with the L^1 norm, as we saw in Section 12.7.

Table 8: Summary of the results for example `TestFOMSearch` with $m = 1048573$ (above) and $m = 1099511627791$ (below).

$m = 1048573$	primal		m-dual	
Method	CPU time	best FOM	CPU time	best FOM
1. BKZ+BB, naive	1989.6	0.237731	1989.2	0.250676
2. BKZ+BB, discard	4.8	0.270944	7.0	0.265395
3. LLL only, discard	4.5	0.270944	6.7	0.265395
4. Two stages, stage 1 with $\boldsymbol{t}$	9.0		10.8	
stage 2 with BKZ+BB	0.02	0.270944	0.02	0.265395
5. Two stages, stage 1 with $\boldsymbol{t}_0$	4.5		6.9	
stage 2 with BKZ+BB	0.06	0.270944	0.02	0.265395

$m = 1099511627791$	primal		m-dual	
Method	CPU time	best FOM	CPU time	best FOM
1. BKZ+BB, naive	2427.2	0.241259	2253.2	0.223998
2. BKZ+BB, discard	7.3	0.264833	9.0	0.269388
3. LLL only, discard	7.0	0.264833	8.6	0.269388
4. Two stages, stage 1 with $\boldsymbol{t}$	13.4		15.1	
stage 2 with BKZ+BB	0.024	0.264833	0.03	0.269388
5. Two stages, stage 1 with $\boldsymbol{t}_0$	7.2		9.6	
stage 2 with BKZ+BB	0.024	0.264833	0.05	0.269388

screening pass with a lower-cost FOM should speed up things, but it does not. The reason is that keeping a list of 50 candidates in the first stage reduces the efficacy of early discarding. It may also be surprising that Method 5 is slower than Method 4. It is probably because Method 4 discards earlier on average. The computing times are generally longer for the m-dual than for the primal. We think this is because computing an m-dual basis for a projection requires more work than just computing a primal basis (which is direct).

Methods 2 to 5 all retained the same best multiplier, with the same FOMs, but the second and third best multipliers and FOMs were not always the same (see the detailed results in the files). Methods 1 returned multipliers of lower quality because it examined fewer candidates. Overall, Methods 4 and 5 perform more poorly than Methods 2 and 3. Method 3 is a bit faster than Method 2, but it does not always return an exact result, because it uses only the LLL approximation, so we should prefer Method 2.

12.12 TestNormBounds

This program tests and compares the normalization constants used in the subclasses of `Normalizer`. These classes use different approximations or bounds for the Hermite constants γ_t, as explained in Sections 9 and 11.8. The program prints these different approximations

for $t = 1, \ldots, 48$, so they can be compared visually. It does the same for the approximations of $\gamma_t^{(1)} = t\gamma_t$ and the values of γ_t^{M} used for the L^1 norm. One can see for example which upper bounds are smallest when we use the L^1 norm. Finally, it also prints the center density values $\delta_t = (\gamma_t/4)^{1/t}$ that correspond to those γ_t approximations, to check if they agree with the values found in the references. We do this because most of our bounds on γ_t were computed from bounds on δ_t given in some references. The resulting tables are in `testNormBounds.res`.

References

[1] L. Afflerbach and H. Grothe. Calculation of Minkowski-reduced lattice bases. *Computing*, 35:269–276, 1985.

[2] L. Afflerbach and H. Grothe. The lattice structure of pseudo-random vectors generated by matrix generators. *Journal of Computational and Applied Mathematics*, 23:127–131, 1988.

[3] W. A. Beyer, R. B. Roof, and D. Williamson. The lattice structure of multiplicative congruential pseudo-random vectors. *Mathematics of Computation*, 25(114):345–363, 1971.

[4] H. F. Blichfeldt. The minimum value of quadratic forms, and the closest packing of spheres. *Mathematische Annalen*, 101(1):605–608, 1929.

[5] R. M. Bremner. *Lattice Basis Reduction: An Introduction to the LLL Algorithm and Its Applications*. Pure and Applied Mathematics. Chapman & Hall, CRC Press, 2012.

[6] H. Cohn. A conceptual breakthrough in sphere packing. *Notices of the American Mathematical Society*, 64(2):102–115, 2017.

[7] H. Cohn and N. Elkies. New upper bounds on sphere packing I. *Annals of Mathematics*, 157(2):689–714, 2003.

[8] J. H. Conway and N. J. A. Sloane. *Sphere Packings, Lattices and Groups*. Grundlehren der Mathematischen Wissenschaften 290. Springer-Verlag, New York, 3rd edition, 1999.

[9] R. Couture and P. L'Ecuyer. Linear recurrences with carry as random number generators. In *Proceedings of the 1995 Winter Simulation Conference*, pages 263–267, 1995.

[10] R. Couture and P. L'Ecuyer. Orbits and lattices for linear random number generators with composite moduli. *Mathematics of Computation*, 65(213):189–201, 1996.

[11] R. R. Coveyou and R. D. MacPherson. Fourier analysis of uniform random number generators. *Journal of the ACM*, 14:100–119, 1967.

[12] J. Dick, P. Kritzer, and F. Pillichshammer. *Lattice Rules: Numerical Integration, Approximation, and Discrepancy*. Springer, 2022.

[13] U. Dieter. How to calculate shortest vectors in a lattice. *Mathematics of Computation*, 29(131):827–833, 1975.

[14] U. Fincke and M. Pohst. Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. *Mathematics of Computation*, 44:463–471, 1985.

[15] G. S. Fishman. *Monte Carlo: Concepts, Algorithms, and Applications*. Springer Series in Operations Research. Springer-Verlag, New York, NY, 1996.

[16] H. Grothe. *Matrixgeneratoren zur Erzeugung Gleichverteilter Pseudozufallsvektoren*. Dissertation (thesis), Tech. Hochschule Darmstadt, Germany, 1988.

[17] P. M. Gruber and C. G. Lekkerkerker. *Geometry of Numbers*. North-Holland, Amsterdam, 1987.

[18] W. Hart, F. Johanssony, and S. Pancratz. *FLINT: Fast Library for Number Theory*, 2015. Reference Manual, Version 2.5.2, available at `www.flintlib.org/flint-2.5.pdf`.

[19] B. Helfrich. Algorithms to construct Minkowski-reduced and Hermite-reduced lattice bases. *Theoretical Computer Science*, 41:125–139, 1985.

[20] F. J. Hickernell, H. S. Hong, P. L'Ecuyer, and C. Lemieux. Extensible lattice sequences for quasi-Monte Carlo quadrature. *SIAM Journal on Scientific Computing*, 22(3):1117–1138, 2001.

[21] E. Hlawka. Zur geometrie der zahlen. *Mathematische Zeitschrift*, 49:285–312, 1943.

[22] J. D. Hobby. A natural lattice basis problem with applications. *Mathematics of Computation*, 67:1149–1161, 1998.

[23] Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC'83)*, pages 193–206, New York, NY, USA, 1983. ACM.

[24] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 12(3):415–440, 1987.

[25] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, third edition, 1998.

[26] J. L. Lagrange. *Recherches d'arithmétique*. Nouveaux mémoires de l'Académie royale des sciences et belles-lettres de Berlin, 1773.

[27] P. L'Ecuyer. Efficient and portable 32-bit random variate generators. In J. R. Wilson, J. O. Henriksen, and S. D. Roberts, editors, *Proceedings of the 1986 Winter Simulation Conference*, pages 275–277, 1986.

[28] P. L'Ecuyer. A portable random number generator for 16-bit computers. In *Modeling and Simulation on Microcomputers 1987*, pages 45–49. The Society for Computer Simulation, 1987.

[29] P. L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31(6):742–749 and 774, 1988. See also the correspondence in the same journal, 32, 8 (1989) 1019–1024.

[30] P. L'Ecuyer. Random numbers for simulation. *Communications of the ACM*, 33(10):85–97, 1990.

[31] P. L'Ecuyer. Combined multiple recursive random number generators. *Operations Research*, 44(5):816–822, 1996.

[32] P. L'Ecuyer. Uniform random number generators. In D. J. Medeiros, E. F. Watson, J. S. Carson, and M. S. Manivannan, editors, *Proceedings of the 1998 Winter Simulation Conference*, pages 97–104. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc., 1998.

[33] P. L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[34] P. L'Ecuyer. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation*, 68(225):249–260, 1999.

[35] P. L'Ecuyer. Uniform random number generation. In S. G. Henderson and B. L. Nelson, editors, *Simulation*, Handbooks in Operations Research and Management Science, pages 55–81. Elsevier, Amsterdam, The Netherlands, 2006. Chapter 3.

[36] P. L'Ecuyer. Quasi-Monte Carlo methods with applications in finance. *Finance and Stochastics*, 13(3):307–349, 2009.

[37] P. L'Ecuyer. Random number generation. In J. E. Gentle, W. Haerdle, and Y. Mori, editors, *Handbook of Computational Statistics*, pages 35–71. Springer-Verlag, Berlin, second edition, 2012.

[38] P. L'Ecuyer. Randomized quasi-Monte Carlo: An introduction for practitioners. In P. W. Glynn and A. B. Owen, editors, *Monte Carlo and Quasi-Monte Carlo Methods: MCQMC 2016*, pages 29–52, Berlin, 2018. Springer.

[39] P. L'Ecuyer. Stochastic simulation and Monte Carlo methods. Draft Textbook, `https://www-labs.iro.umontreal.ca/~lecuyer/ift6561/book.pdf`, 2023.

[40] P. L'Ecuyer. Latmrg user's guide. `https://www-labs.iro.umontreal.ca/~lecuyer/papers.html` (forthcoming), 2025.

[41] P. L'Ecuyer and T. H. Andres. A random number generator based on the combination of four LCGs. *Mathematics and Computers in Simulation*, 44:99–107, 1997.

[42] P. L'Ecuyer and F. Blouin. BonGCL, un lociciel pour la recherche de bons générateurs à congruence linéaire. Technical Report DIUL-RT-8803, Compter Science Department, Laval University, Ste-Foy (Que.), Canada, 1988.

[43] P. L'Ecuyer and F. Blouin. Linear congruential generators of order $k > 1$. In *Proceedings of the 1988 Winter Simulation Conference*, pages 432–439. IEEE Press, 1988.

[44] P. L'Ecuyer, F. Blouin, and R. Couture. A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation*, 3(2):87–98, 1993.

[45] P. L'Ecuyer, E. Bourceret, D. Munger, M.-A. Savard, R. Simard, M. Thiongane, and C. Weiss. Lattice Tester. `https://github.com/pierrelecuyer/latticetester`, 2025.

[46] P. L'Ecuyer, E. Bourceret, D. Munger, M.-A. Savard, R. Simard, and P. Wambergue. Latmrg. `https://github.com/umontreal-simul/LatMRG`, 2022.

[47] P. L'Ecuyer and R. Couture. An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing*, 9(2):206–217, 1997.

[48] P. L'Ecuyer and R. Couture. *LatMRG User's Guide: A Modula-2 software for the theoretical analysis of linear congruential and multiple recursive random number generators*, 2000. `http://www.iro.umontreal.ca/~lecuyer/myftp/papers/guide-latmrg-m2.pdf`.

[49] P. L'Ecuyer, M. Godin, A. Jemel, P. Marion, and D. Munger. LatNet Builder: A general software tool for constructing highly uniform point sets. `https://github.com/umontreal-simul/latnetbuilder`, 2019.

[50] P. L'Ecuyer and C. Lemieux. Variance reduction via lattice rules. *Management Science*, 46(9):1214–1235, 2000.

[51] P. L'Ecuyer, P. Marion, M. Godin, and F. Puchhammer. A tool for custom construction of QMC and RQMC point sets. In A. Keller, editor, *Monte Carlo and Quasi-Monte Carlo Methods: MCQMC 2020*, pages 51–70, Berlin, 2022. Springer.

[52] P. L'Ecuyer and D. Munger. On figures of merit for randomly-shifted lattice rules. In H. Woźniakowski and L. Plaskota, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2010*, pages 133–159, Berlin, 2012. Springer-Verlag.

[53] P. L'Ecuyer and D. Munger. Algorithm 958: Lattice builder: A general software tool for constructing rank-1 lattice rules. *ACM Transactions on Mathematical Software*, 42(2):Article 15, 2016.

[54] P. L'Ecuyer, O. Nadeau-Chamard, Y.-F. Chen, and J. Lebar. Multiple streams with recurrence-based, counter-based, and splittable random number generators. In *Proceedings of the 2021 Winter Simulation Conference*, pages 1–16. IEEE Press, 2021.

[55] P. L'Ecuyer, G. Perron, and F. Blouin. SENTIERS: Un logiciel Modula-2 pour l'arithmétique sur les grands entiers. Technical Report DIUL-RT-8802, Computer Science Department, Laval University, 1988.

[56] P. L'Ecuyer and R. Simard. Beware of linear congruential generators with multipliers of the form $a = \pm 2^q \pm 2^r$. *ACM Transactions on Mathematical Software*, 25(3):367–374, 1999.

[57] P. L'Ecuyer and R. Simard. On the lattice structure of a special class of multiple recursive random number generators. *INFORMS Journal on Computing*, 26(2):449–460, 2014.

[58] P. L'Ecuyer and S. Tezuka. Structural properties for two classes of combined random number generators. *Mathematics of Computation*, 57(196):735–746, 1991.

[59] P. L'Ecuyer and R. Touzin. Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$. In *Proceedings of the 2000 Winter Simulation Conference*, pages 683–689. IEEE Press, 2000.

[60] P. L'Ecuyer and R. Touzin. On the Deng-Lin random number generators and related methods. *Statistics and Computing*, 14:5–9, 2004.

[61] P. L'Ecuyer, P. Wambergue, and E. Bourceret. Spectral analysis of the MIXMAX random number generators. *INFORMS Journal on Computing*, 32(1):135–144, 2020.

[62] C. Lemieux. *Monte Carlo and Quasi-Monte Carlo Sampling*. Springer-Verlag, 2009.

[63] C. Lemieux and P. L'Ecuyer. A comparison of Monte Carlo, lattice rules and other low-discrepancy point sets. In H. Niederreiter and J. Spanier, editors, *Monte Carlo and Quasi-Monte Carlo Methods 1998*, pages 326–340, Berlin, 2000. Springer-Verlag.

[64] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

[65] L. Lovász. *An Algorithmic Theory of Numbers, Graphs and Convexity*. Number 50 in SIAM CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, Philadelphia, 1986.

[66] Leon Mächler and David Naccache. A conjecture on Hermite constants. Cryptology ePrint Archive, Paper 2022/677, 2022.

[67] G. Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences of the United States of America*, 60:25–28, 1968.

[68] A. Neumaier. Bounding basis reduction properties. *Designs, Codes and Cryptography*, 84(1):237–259, 2017.

[69] P. Q. Nguyen. Hermite constants and lattice algorithms. In Phong Q. Nguyen and Brigitte Vallée, editors, *The LLL Algorithm: Survey and Applications*, pages 19–69. Springer Verlag, Berlin, Heidelberg, 2010.

[70] P. Q. Nguyen and B. Vallée, editors. *The LLL Algorithm: Survey and Applications*. Springer Verlag, Berlin, Heidelberg, 2010.

[71] M. Pohst. On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. *ACM SIGSAM Bulletin*, 15:37–44, 1981.

[72] M. Pohst. A modification of the LLL reduction algorithm. *Journal of Symbolic Computation*, 4:123–127, 1987.

[73] C. A. Rogers. The number of lattice points in a set. *Proceedings of the London Mathematical Society*, s3-6(2):305–320, 1956.

[74] C. A. Rogers. Lattice coverings of space. *Mathematika*, 6(1):33–39, 1959.

[75] M.-A. Savard. Générateurs de nombres aléatoires modulo un grand entier, dont l'uniformité est assurée. Master's thesis, DIRO, Université de Montréal, 2020. `https://www-labs.iro.umontreal.ca/~lecuyer/myftp/theses/msc-thesis-savard2020.pdf`.

[76] C. P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53(2):201–224, 1987.

[77] C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In L. Budach, editor, *Fundamentals of Computation Theory: 8th International Conference*, pages 68–85, Berlin, Heidelberg, 1991. Springer-Verlag.

[78] Claus Peter Schnorr. Progress on LLL and lattice reduction. In Phong Q. Nguyen and Brigitte Vallée, editors, *The LLL Algorithm: Survey and Applications*, pages 145–178. Springer Verlag, Berlin, Heidelberg, 2010.

[79] V. Shoup. *NTL: A Library for doing Number Theory*. Courant Institute, New York University, New York, NY, July 2018. `https://shoup.net/ntl/`.

[80] Carl Ludwig Siegel. A mean value theorem in geometry of numbers. *Annals of Mathematics*, 46(2):340–347, 1945.

[81] I. H. Sloan and S. Joe. *Lattice Methods for Multiple Integration*. Clarendon Press, Oxford, 1994.

[82] Arne Storjohann. Faster algorithms for integer lattice basis reduction. Technical report, Department Informatik, ETH Zurich, 1996. Technical Report 249.

[83] D. S. Watkins. *Fundamentals of Matrix Computation*. Wiley, second edition, 2002.

[84] Jinming Wen and Xiao-Wen Chang. Sharper bounds on four lattice constants. *Designs, Codes and Cryptography*, 90:1463–1484, 2022.

[85] Jinming Wen, Xiao-Wen Chang, and Jian Weng. Improved upper bounds on the Hermite and KZ constants. In *2019 IEEE International Symposium on Information Theory (ISIT)*, pages 1742–1746, 2019.