Lattice Tester:
A Software Tool to Analyze Integral Lattices

Pierre L'Ecuyer[1] and Christian Weiß[2]

[1] DIRO, Université de Montréal, Canada,
lecuyer@iro.umontreal.ca,
https://www-labs.iro.umontreal.ca/~lecuyer/
[2] Ruhr West University of Applied Sciences, Germany,
christian.weiss@hs-ruhrwest.de,
https://www.hochschule-ruhr-west.de/personenseiten/christian-weiss

Abstract. Lattice Tester is a C++ software library offering tools to measure the uniformity of lattices in the t-dimensional integer space $\mathbb{Z}^t$. Such lattices may come from quasi-Monte Carlo point sets or from multiple recursive linear random number generators, for example. The uniformity measures include the length of a shortest nonzero vector in the lattice or in its dual (the spectral test), and figures of merit that use normalized versions of these lengths for projections of the lattice on large subsets of coordinates, and take the worst-case over the considered projections. This paper gives a brief tour of Lattice Tester, its algorithms, its design, and numerical illustrations. More details and examples can be found in the user's guide and the code is freely available on GitHub.

Keywords: lattice structure, spectral test, random number generators

1 Introduction and Overview

The set of vectors of t successive outputs from a linear *random number generator* (RNG) in scalar or matrix form, from all possible initial states, is known to be the intersection of a lattice L_t with the t-dimensional unit hypercube $[0, 1)^t$. This holds in particular for linear congruential generators (LCGs), multiple recursive generators (MRGs), multiply-with-carry generators, and combinations of them [4,6,16,17,19,24,32,35]. Measuring the uniformity of this lattice for a range of values of t and also of its projections over subsets of coordinates is a key step in the theoretical analysis of such RNGs. The main measures that we want to compute for these lattices are the lengths of shortest non-zero vectors in the primal lattice L_t or its dual lattice L_t^*, for the L^2 and L^1 norms. With the Euclidean norm, the inverse of the length of a shortest nonzero vector in L_t^* 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 as small as possible for the points to cover the space evenly, i.e., the shortest length should be large. With the L^1 norm, this length corresponds to the minimal number of hyperplanes that contain all the lattice points in the unit cube. We also want it to be large. Note that all integer multiples of any lattice vector are in the lattice. Thus, if there is a very short nonzero vector in the primal lattice, its multiples form a line of equidistant lattice points that

are very close to each other, which implies that the lattice points must be concentrated on fewer lines, which is bad. So we also want the shortest nonzero vector in the primal lattice to be large.

A small illustration. To illustrate this, Figure 1 shows all the pairs (u_{n-1}, u_n) of successive outputs of the tiny linear congruential generator (LCG) defined by the recurrence $x_n = ax_{n-1} \bmod m$ and $u_n = x_n/m$, for $m = 101$ with $a = 12$ (left) and $a = 51$ (right). These 101 points have an obvious lattice structure. Each blue point can be expressed as an integer linear combination of the two red vectors, $\boldsymbol{v}_1 = (1/101,\ a/101)$ and $\boldsymbol{v}_2 = (0, 1)$, which form a basis of the lattice. The lattice contains all integer linear combinations of the basis vectors, so it has an infinite number of points. The lattice points in the picture are those that belong to the square $[0, 1)^2$. On the right, the shortest nonzero vector is small and the distance between the hyperplanes (the lines in this case) that contain all the points is very large, so we have poor uniformity, whereas on the left they cover the square more evenly.

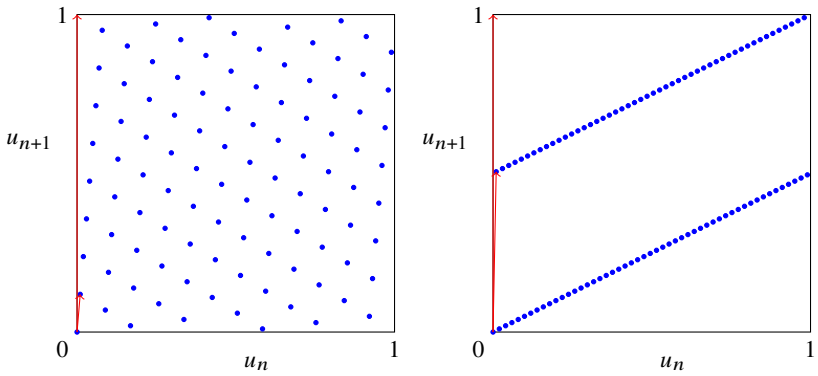

Fig. 1. Lattice structure for LCGs with $m = 101$, for $a = 12$ (left) and $a = 51$ (right).

Integral lattices and measures of uniformity. For general background on lattices in the real space, see for example [2,3,40]. Computing the length of a shortest nonzero vector in the dual lattice is known as the *spectral test* [7,14,17,27].

Similar lattices are used for *lattice rules*, which are quasi-Monte Carlo integration methods that estimate the integral of a function over the unit hypercube by its average over a set of lattice points over the unit hypercube [8,20,22,28,30,45]. Bounds on the worst-case integration error for certain classes of functions can be obtained in terms of *figures of merit* (FOMs) that measure the uniformity of the lattice. Measuring the uniformity is also essential in that setting. The main difference between these lattices and those encountered in RNGs is that the number η of lattice points per unit of volume, called the lattice density, is usually modest (at most in the millions) for lattice rules, but

much larger (more than 2^{100}) 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 [8,20]) correspond to integration error bounds for certain classes of functions. They require a computing time that increases at least linearly with η and are unusable when $n \geq 2^{100}$. The lattice structure of RNGs is usually analyzed via the spectral test, whose computation requires exponential times in t in the worst case but only logarithmic time in η, so very large values of η can be handled. The spectral test is also relevant and has been used in FOMs for lattice rules [13,28,36,37].

The aim of this paper is to describe Lattice Tester, a software library written in C++ that offers tools to compute FOMs that serve as measures of uniformity for t-dimensional *integral lattices*, i.e., for which all lattice point coordinates are integers. In most cases of interest, this integrality property already holds for the dual lattice, and it is obtained for the primal lattice by rescaling all vector coordinates by an integer factor $m > 1$ that typically corresponds to the modulus of the generator. In the small example seen earlier, we would do this with $m = 101$. Then we work with the rescaled lattice $\Lambda_t = mL_t$ instead of the original one. The purpose of this rescaling is for all vector coordinates to be integers, so they can always be represented exactly on the computer. The dual of the original lattice is then the m-dual of the rescaled one.

What Lattice Tester does. The main facilities provided by Lattice Tester are the following:

A. *Lattice Basis Construction*. Given a set of vectors with integer coordinates, find a basis for the lattice generated by these vectors.
B. *Find the 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.
D. *Shortest Vector Problem (SVP).* Find a (provably) shortest nonzero lattice vector.
E. *Figure of Merit Calculation.* Compute a FOM that takes into account several lower-dimensional projections of the lattice.

Problems A, B, and C are relatively easy, whereas D is harder. The NTL library [44] already implements the LLL and BKZ procedures for Problem C, and we use these implementations. For A and B, we have faster algorithms than what we have seen before. For D, we use a *branch-and-bound* (BB) integer programming optimization procedure based on the algorithm of [10], with some modifications. It is implemented for both the L^2 and L^1 norms. It uses a Cholesky decomposition of a previously reduced basis, and is much faster than the old spectral test algorithms given in [9,14]. We also examine a variant that uses a triangular basis to compute the bounds.

The latest version of Lattice Tester is built largely over NTL [44], which offers arbitrarily large integers and real numbers via the types `NTL::ZZ` and `NTL::RR`, polynomial and modular arithmetic, and many useful tools. Lattice Tester has flexible types named `Int` for integers and `Real` for real numbers. `Int` can be either `int64_t` or `NTL::ZZ`, whereas `Real` can be `double`, `xdouble`, `quad_float`, or `NTL::RR` [23]. The desired types can be selected via template parameters used by the compiler to compile the code with the types we want, so very large values of m can be handled with `NTL::RR` if

needed while the same code can run much faster by using `double` when this is sufficient. `NTL::RR` can handle large real numbers with arbitrary precision, but make the computations much slower, often more than 50 times slower than using `double`, in our experiments.

Lattice Tester is implemented as a library meant to be used by other software. Its previous version is currently used by LatMRG [26] and by LatNet Builder [29], which provide tools to construct lattice bases for specific applications. It also contains executable programs that can serve as examples, including those we used for the experiments reported in this paper. The software is available on GitHub [25] and a detailed user's guide can be found in [23].

History. The main facilities offered by Lattice Tester were included in the old LatMRG software [27], whose first version was written around 1988 in the Modula-2 language, and whose aim was to study the lattice structure of linear RNGs and search for good ones. That Modula-2 version was used for [16,17,18,31,33,34], for example. It incorporated the BB algorithm from [10], 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. It used a home-made Modula-2 library for the arithmetic with arbitrarily large integers. After 2000, Modula-2 was no longer supported, and some students and assistants started "translating" LatMRG into C++ and making changes and extensions. This went on for about 15 years without reaching a final product. In 2014, David Munger separated LatMRG in two pieces: (1) Latcommon, which later became Lattice Tester, whose task is to compute measures of uniformity for arbitrary integer lattices, and (2) a new LatMRG which uses those facilities to test the lattice structure of RNGs and search for good generators under various types of constraints. The rationale for this split was to make Lattice Tester smaller and more easily usable to analyze lattices for other purposes than testing RNGs. Further changes were made by various people after 2014. This evolution led to a code that was complicated, inconsistent, inefficient in some places, and poorly documented. Recently, we decided to redesign and recode Lattice Tester and LatMRG. Some new algorithms also came out of this. The aim of this paper is to present the new Lattice Tester. Another (forthcoming) paper will present the new LatMRG.

Outline. The remainder is organized as follows. In Section 2, we define the integral lattices considered here, their m-dual, and we recall some properties. Section 3 explains how to obtain a lattice basis from a set of generating vectors and compute its m-dual. Section 4 outlines our BB procedure to compute a shortest nonzero vector in the lattice. In Section 5, we briefly recall the notions of LLL and BKZ reduction, which are the main ones used in the software. In Section 6, we define FOMs that the software can compute. In Section 7, we report numerical results that provide insight on how the algorithms and the software perform in practice. Section 8 provides a conclusion.

2 Integral Lattices

The *integral lattices* considered here have the form

$$\Lambda_t = \left\{ \boldsymbol{v} = \boldsymbol{z}\boldsymbol{V} = \sum_{j=1}^{t} z_j \boldsymbol{v}_j \ \text{ such that } \ \boldsymbol{z} = (z_1, \ldots, z_t) \in \mathbb{Z}^t \right\}, \tag{1}$$

where t is a positive integer and $\boldsymbol{V}$ is a matrix whose rows $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ are linearly independent vectors in $\mathbb{Z}^t$ which form a *basis* of the lattice. The *density* of Λ_t, which is the average number of lattice points per unit of volume, is $1/\det(\boldsymbol{V}) = \det(\boldsymbol{V}^{-1})$. When $\Lambda_t = mL_t$ is a rescaled version of a (non-integral) lattice L_t in the real space, $\eta_t = m^t/\det(\boldsymbol{V})$ is the density of L_t, and the cardinality of the point set $L_t \cap [0,1)^t$. For any subset of coordinates $I = \{i_1, \ldots, i_s\} \subseteq \{1, \ldots, t\}$, the *projection* Λ_I of Λ_t over the s-dimensional subspace determined by I is also an integral lattice

Let $\boldsymbol{v} \cdot \boldsymbol{w}$ denote the standard inner product of vectors $\boldsymbol{v}$ and $\boldsymbol{w}$ (for the L^2 norm). The *m-dual lattice* of Λ_t is $\Lambda_t^* = \{\boldsymbol{h} \in \mathbb{R}^t \mid \boldsymbol{h} \cdot \boldsymbol{v} \in m\mathbb{Z} \text{ for all } \boldsymbol{v} \in \Lambda_t\}$. It is assumed that $\Lambda_t^* \subset \mathbb{Z}^t$, so it is also an integral lattice. The *m-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 $\mathbb{Z}^t$ such that $\boldsymbol{v}_i \cdot \boldsymbol{w}_j = m\delta_{ij}$, where $\delta_{ij} = 1$ if $i = j$, and $\delta_{ij} = 0$ otherwise. The vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_t$ are a basis of the m-dual lattice. They are the rows of a matrix $\boldsymbol{W} = m(\boldsymbol{V}^{-1})^{\mathrm{t}}$. The density η_t^* of Λ_t^* is $\eta_t^* = 1/\eta_t$.

Since $\boldsymbol{V} \cdot \boldsymbol{W}^{\mathrm{t}} = m\boldsymbol{I}$ and all the entries of $\boldsymbol{V}$ and $\boldsymbol{W}$ are integers, each vector $m\boldsymbol{e}_i$ (m times the ith unit vector) can be expressed as an integer linear combination of the rows of $\boldsymbol{V}$ and also the rows of $\boldsymbol{W}$, so it must belong to both the primal and m-dual lattices. This implies that all operations on vectors to construct or reduce a basis can be performed modulo m, as long as this does not exclude a vector $m\boldsymbol{e}_i$ from the lattice.

3 Basis Construction

Often, a lattice basis is given directly from the problem specification. This is the case in particular when the lattice corresponds to successive output values of a linear RNG [14,15,27]. But sometimes, we need to construct a basis for the lattice generated by a finite set of vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s \in \mathbb{Z}^t$ together with the vectors $m\boldsymbol{e}_1, \ldots, m\boldsymbol{e}_t$. We may also need the corresponding m-dual basis. This occurs in particular when we want a basis for the projection Λ_I of Λ_t over a subset I of coordinates, and for its m-dual Λ_I^*. The vectors $m\boldsymbol{e}_i$ always belong to the lattice and we always add them to the set of generating vectors, often implicitly. Then the generated lattice always has t dimensions.

Our fastest construction method returns either a lower-triangular or an upper-triangular $t \times t$ matrix basis. For the upper-triangular case, it works as follows. We start with $s \geq 1$ vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s$, with $\boldsymbol{v}_i = (v_{i,1}, \ldots, v_{i,t})$. We first replace each $v_{i,j}$ by $v_{i,j} \bmod m$, to obtain a set of s generating vectors with $0 \leq v_{i,j} < m$ for all (i, j), together with the vectors $m\boldsymbol{e}_i$. If $v_{i,1} = 0$ for all i, we put $c_1 = m$, and we take $\boldsymbol{x}_1 = m\boldsymbol{e}_1$ as our first basis vector. Otherwise, let $c_1 = \gcd(v_{1,1}, \ldots, v_{s,1}, m)$, assuming here that the zero values are skipped. When computing c_1 with Euclid's algorithm, we obtain $c_1 = a_{0,1}m + a_{1,1}v_{1,1} + \cdots + a_{s,1}v_{s,1}$ for some integers $a_{i,1}$ with $-m < a_{i,1} < m$. Our first triangular basis vector is $\boldsymbol{x}_1 = a_{1,1}\boldsymbol{v}_1 + \cdots + a_{s,1}\boldsymbol{v}_s \bmod m$ and it has $c_1 > 0$ as its first

coordinate. Since 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 zero as their first coordinate, and their other coordinates are in $\mathbb{Z}_m$. 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. Then we repeat the same process with the new vectors $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_s$, but using their second column, then with the third column, and so on, until we have $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_t$ which form an upper-triangular basis.

A similar algorithm briefly described in Section 7 of [5] and in Section 3 of [27] was implemented in the Modula-2 version of LatMRG. That algorithm was modifying the vectors $\boldsymbol{x}_i$ along the way while computing the gcd. The algorithm described here is about 5 times faster in our tests.

For a lower-triangular basis, we proceed in a symmetric way by first computing $c_t = \gcd(v_{1,t}, \ldots, v_{s,t}, m) = a_{0,t}m + a_{1,t}v_{1,t} + \cdots + a_{s,t}v_{s,t}$ for the last column, putting either $\boldsymbol{x}_t = m\boldsymbol{e}_t$ or $\boldsymbol{x}_t = a_{1,t}\boldsymbol{v}_1 + \cdots + a_{s,t}\boldsymbol{v}_s \bmod m$ as our last basis vector, and so on.

Once we have a lattice basis $\boldsymbol{V}$, computing its m-dual $\boldsymbol{W}$ means solving the linear system $\boldsymbol{V}\boldsymbol{W}^{\mathrm{t}} = m\boldsymbol{I}$. We have a general implementation of that based on some NTL facilities. However, this is generally much too slow unless the dimension is very small. Computing the m-dual is much easier and faster when the basis is triangular. When $\boldsymbol{V}$ is upper-triangular, its m-dual $\boldsymbol{W}$ must be lower-triangular and its entries are given directly by $w_{i,i} = m/v_{i,i}$ for $i = 1, \ldots, t$ and $w_{i,j} = -\frac{1}{v_{j,j}} \sum_{k=j+1}^{i} v_{k,j} w_{i,k}$ for $1 \le j < i \le t$. All these entries are integer and can be computed exactly using integer arithmetic. It is important to recall that the m-dual of a projection is not the same as the projection of the m-dual over the same coordinate set I. Concrete examples are given in [23]. To examine the m-dual of a projection, we need to construct a basis for the projection and then compute its m-dual.

To construct a basis from a set of generating vectors, NTL applies an LLL reduction to this set of vectors. The result is a set of t vectors that form a basis, plus some zero vectors. One advantage of this method is that the basis vectors are reduced, so they tend to be shorter than with the triangular method. But the triangular method is faster, and if an m-dual basis is also required, it can be computed much more efficiently from a triangular basis.

4 Shortest Vector Problem

A key task of Lattice Tester is to compute a shortest nonzero vector in a lattice. Here we assume that the lattice basis is given by the rows $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ of the matrix $\boldsymbol{V}$. The same can be done with the m-dual instead of the primal lattice.

Any lattice vector $\boldsymbol{v}$ must be an integer linear combination of the basis vectors, so finding a shortest nonzero vector for the L^p norm can be formulated as the following integer programming optimization problem with decision variables $z_1, \ldots, z_t$:

$$\text{Minimize} \quad \|\boldsymbol{v}\|_p \qquad \text{subject to } \boldsymbol{v} = \boldsymbol{z}\boldsymbol{V} = \sum_{i=1}^{t} z_i \boldsymbol{v}_i, \quad z_i \in \mathbb{Z}, \quad \sum_{i=1}^{t} |z_i| > 0. \tag{2}$$

In the following, we denote by $b(p)$ the L^p norm of the shortest currently-known vector in this lattice. This vector may not be in the basis, but $b(p)$ is an upper bound on the

length of a shortest vector, and we are only interested in vectors shorter than that. As in [10], we solve this problem via a *branch-and-bound* (BB) algorithm in which 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 critical part of the algorithm is how we obtain these bounds. We examine and compare two ways of doing that: (1) using a triangular basis and (2) via a Cholesky decomposition of the matrix of scalar products of basis vectors.

Supose $\boldsymbol{V} = \boldsymbol{L}$ is *lower-triangular* with elements $\ell_{i,j}$. For $\boldsymbol{v} = \boldsymbol{z}\boldsymbol{L}$, we have

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

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

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

For $\boldsymbol{v}$ to be shorter than our current best, we must have $s_{j-1}(p) < b(p)^p$, which implies

$$z_j^{\min} = z_j^{\min}(p) := \lceil c_j - \delta_j(p) \rceil \;\leq z_j \leq\; \lfloor c_j + \delta_j(p) \rfloor =: z_j^{\max}(p) = z_j^{\max}, \qquad (3)$$

where $c_j = -r_j/\ell_{j,j}$ is the center of the interval and $\delta_j(p) = (b(p)^p - s_j(p))^{1/p}/\ell_{j,j}$ is the radius when positive (otherwise the interval is empty).

The BB algorithm then works as follows. We start by computing the bounds (3) for $j = t$. For each value of z_j in the interval, we compute the bounds on z_{j-1} when $z_j, \ldots, z_t$ are fixed and examine each value of z_{j-1} in the interval. We do this recursively for $j = t-1, \ldots, 0$. When $j = 0$, we are at a leaf of the BB tree, with $z_t, \ldots, z_1$ all fixed. If $\boldsymbol{z} \neq \boldsymbol{0}$, we check if the length of the corresponding vector $\boldsymbol{z}\boldsymbol{V}$ is shorter than our best one, in which case we save it. We repeat this until all tree nodes have been explored. The Lattice Tester guide [23] provides additional explanations and implementation details.

By looking at the formula for $\delta_j(p)$, we see that we want $b(p)$ as small as possible and $\ell_{j,j} > 0$ as large as possible, to reduce the width of the interval (3). But since $1/\prod_{j=1}^{t} \ell_{j,j} = 1/\det(\boldsymbol{V})$ is the density of the lattice and is fixed, it is not possible to increase all the $\ell_{j,j}$'s. Typically, the projection of Λ_t over any single coordinate contains all the integers, which implies that $\ell_{t,t} = 1$ for any lower-triangular basis, and then the number of values of z_t to examine (at the first level of the tree) will always be very large, which limits the efficiency of using a triangular basis to compute the bounds in the BB algorithm. Numerical examples of this are given in [23].

Another way of getting bounds when $p = 2$ (the Euclidean norm) works as follows [1,10,41]. For an arbitrary basis $\boldsymbol{V}$, let $\boldsymbol{V}\boldsymbol{V}^{\mathrm{t}} = \boldsymbol{L}\boldsymbol{L}^{\mathrm{t}}$ where $\boldsymbol{L}$ is a lower-triangular matrix with elements $\ell_{i,j}$. This is the Cholesky decomposition of the matrix $\boldsymbol{V}\boldsymbol{V}^{\mathrm{t}}$ of inner products of the basis vectors, with the L^2 norm. For any lattice vector $\boldsymbol{v} = \boldsymbol{z}\boldsymbol{V}$, using the same notation as in (3), we can write

$$\|\boldsymbol{v}\|_2^2 = \boldsymbol{z}\boldsymbol{V}\boldsymbol{V}^{\mathrm{t}}\boldsymbol{z}^{\mathrm{t}} = \boldsymbol{z}\boldsymbol{L}\boldsymbol{L}^{\mathrm{t}}\boldsymbol{z}^{\mathrm{t}} = \sum_{k=1}^{t} v_k^2 \;\geq\; s_{j-1}(2) \;=\; |z_j \ell_{j,j} + r_j|^2 + s_j(2). \qquad (4)$$

This gives the same bounds as in (3), valid for $p = 2$.

This can be extended to bounds for $0 < p < 2$ by noting that $\|\boldsymbol{v}\|_2 \leq \|\boldsymbol{v}\|_p$ in that case. We can use this to show that the bounds (3) are also valid for $0 < p < 2$ if we take $\delta_j(p) = (b(p)^2 - s_j(2))^{1/2}/\ell_{j,j}$. The details are in [23].

5 Basis Reduction

An important first step before computing a shortest nonzero vector in a lattice is to reduce the basis to make its vectors as short and orthogonal as we can in reasonable time. In the real space $\mathbb{R}^t$, a basis can be made orthogonal (exactly) via the Gram-Schmidt orthogonalization (GSO) process, which modifies basis vectors by adding linear combinations (with real coefficients) of other basis vectors. The basis vectors can also be made as short as we want. But for a lattice basis, we can only add linear combinations with integer coefficients, and the basis vectors cannot be made as short as we want. One of the many definitions of "reduced basis" in that setting, is the following. A basis $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_t$ is called *Hermite-Korkine-Zolotarev (HKZ)-reduced* [2] if: (i) $\boldsymbol{v}_1$ is a shortest nonzero lattice vector; (ii) one cannot reduce another basis vector by adding an integer multiple of $\boldsymbol{v}_1$; (iii) the projection of $\boldsymbol{v}_2, \ldots, \boldsymbol{v}_t$ on the subspace that is orthogonal to $\boldsymbol{v}_1$ is HKZ-reduced. Algorithms to compute a HKZ-reduced basis can be found in [2, Chapter 11], but their running time is very long and exponential in t.

A simple reduction method proposed and used in [9,14] is *pairwise reduction*, where at each step we try to reduce the length of one basis vector by adding an integer multiple of another basis vector, and we stop when it is no longer possible to reduce the length of any basis vector by adding an integer multiple of another basis vector. This is easy to achieve but not sufficient for what we need for the BB algorithm.

The *LLL reduction* algorithm proposed in [38] is weaker than HKZ, but much stronger than pairwise reduction. It provides in polynomial time a set of nearly orthogonal basis vectors by approximating in some sense the GSO process. The returned basis is called *LLL reduced with factor* δ where $1/4 < \delta < 1$ and δ is a parameter to specify. For the details, see [23,38,40,43]. A larger δ gives better reduction but takes more time. Some authors fix δ at 3/4 [12,39,40], but we prefer values closer to 1. Then, the returned $\boldsymbol{v}_1$ is typically not much longer than a shortest nonzero vector in the lattice.

A stronger reduction is the *blockwise Korkine-Zolotarev* (BKZ) reduction introduced in [42], which essentially strengthens LLL by putting additional conditions that depend on another parameter k called the *block size*. BKZ can be seen as an approximation of the HKZ reduction. The larger is k, the stronger is the reduction. With $k = 2$ it is equivalent to LLL-reduction. See [43] for an algorithm and further details. For $1/4 < \delta < 1$ and $k \geq 2$, the algorithm returns a basis that is *BKZ-reduced with factor* δ *for block sizes* k.

The LLL and BKZ are implemented in NTL (for the L^2 norm only) and we use these implementations with minor modifications. LLL and BKZ do not always return a shortest vector; this is why we also need the BB algorithm.

6 Figures of Merit That Examine Projections

When comparing generators, we look at shortest vector lengths for several values of t and also for projections over subsets I of non-successive coordinates. These lengths are normalized to values between 0 and 1: we divide the raw values by upper bounds on the best possible value that can be achieved for the given lattice density and dimension. In our setting, we assume that there is an integer $k \geq 1$ for which the density of Λ_I is 1 for $s \leq k$ and m^{k-s} for $s > k$, while the density of Λ_I^* is m^{-s} for $s \leq k$ and m^{-k} for $s > k$. (Do not confound this k with the BKZ block size defined earlier, both are standard notation.) This assumption holds for the lattices that we are interested in. In particular, for rank-1 lattice rules, we have $k = 1$, while for MRGs, k corresponds to the order of the recurrence [27]. There are several ways of selecting the upper bounds; see [23] for the details.

To define a *figure of merit* (FOM) for the lattice, we select a class $\mathcal{I}$ of subsets of coordinate indices $I = \{i_1, \ldots, i_s\} \subseteq \{1, \ldots, t\}$, compute the normalized shortest vector length for each one, perhaps divide it by a weight ω_I that depends on I, and take either the worst case (minimum) or the sum of these weighted normalized values as the FOM. This can be done for either the primal or m-dual lattice, with either the L^1 or L^2 norm.

As a concrete example, a standard FOM that is implemented takes $\mathcal{I} = S_1(t_1) \cup S_2(t_2) \cup \cdots \cup S_d(t_d)$ for some integer $d \geq 1$ and vector of integers $\boldsymbol{t} = (t_1, \ldots, t_d) \geq \mathbf{0}$, where $S_1(t_1) = \{I = \{1, \ldots, s\} \mid d + 1 \leq s \leq t_1\}$ and $S_s(t_s) = \{I = \{i_1, \ldots, i_s\} \mid 1 \leq i_1 < \cdots < i_s \leq t_s\}$ for $s = 2, \ldots, d$. The worst-case FOM is then defined as

$$M_{\boldsymbol{t}} = M_{t_1,\ldots,t_d} = \min_{1 \leq s \leq d} \min_{I \in S_s(t_s)} \frac{\ell_I}{\omega_I \, \tilde{\ell}_s^*(m,k)} \tag{5}$$

where $s = |I|$ and $\tilde{\ell}_s^*(m,k)$ is the normalization upper bound for the m-dual. 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 giving a smaller weight ω_I to a projection I reduces its importance because the minimum in (5) is then less likely to reached by this projection. This type of FOM was used in [19,21,28], for example. The special case with $d = 1$ and unit weights has been used for selecting LCGs and MRGs [11,17,18,24].

When the lattice comes from a linear RNG based on a recurrence, adding the same integer to all the indexes in the set I does not change the point set. Then, imposing the extra condition $i_1 = 1$ in the definition of $S_s(t_s)$ does not change the value of $M_{t_1,\ldots,t_d}$ and reduces the number of sets I to consider. In our implementation, the user has the choice of imposing this condition or not. We write the FOM $M_{\boldsymbol{t}}^{(1)}$ when the condition is imposed. In that case, the new set $S_s(t_s)$ has cardinality $\binom{t_s-1}{s-1}$ and $M_{\boldsymbol{t}}^{(1)}$ 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 when the condition $i_1 = 1$ is not imposed. For $d = 4$ and $t_s = 32$ for all s, for example, we have $5,019$ projections with the condition compared to $41,444$ without it.

When searching for a lattice with the best possible FOM in a given class, it is not necessary to evaluate all the terms of the FOM for each candidate. We usually keep a minimal passing target and discard a candidate as soon as we know its FOM will be below the target. We call this *early discard*. In case we want to find the r best candidates,

the target will usually be the FOM of the rth best retained candidate so far. The order in which we evaluate the terms in (5) also has an impact on the speed of the search. It is usually faster to evaluate the low-order terms first: start with the projections of $S_2(t_2)$, then $S_3(t_3)$, etc., and end with $S_1(t_1)$.

One may also consider a two-stage strategy for the search. In the first stage, we may use just a fast evaluation (e.g., only LLL and perhaps a vector $\boldsymbol{t}$ with smaller coordinates) to preselect a list of good candidates. In the second stage, we would further test the retained candidates with a more complete evaluation including BKZ+BB, to select the best ones. We may also consider more than two stages, narrowing the list of candidates at each stage. At first, we thought that this would be the most effective approach, but we found out in our experiments that a single stage is usually as good or better. The reason is that maintaining a larger list in the first stage makes the early discard less effective by reducing the passing target.

7 Experimental Results with Lattice Tester

We report a few results from experiments made to compare underlying algorithms, evaluation methods, and search strategies, for lattices that correspond to LCGs with with modulus m and multiplier a, similar to the small example of the introduction. In each case, an initial basis $\boldsymbol{V}_0$ for the rescaled lattice is obtained by taking $\boldsymbol{v}_1 = (1, a, a^2 \bmod m, \ldots, a^{t-1} \bmod m)$ and $\boldsymbol{v}_i = m\boldsymbol{e}_i$ for $i = 2, \ldots, t$. For all the reported results, $m = 1099511627791$, a prime integer near 2^{40}, and the flexible types are `Int = NTL::ZZ, Real = double`. The computations were made on a Intel Core i9-12900H processor running Ubuntu. The programs used for these experiments (and many more) are included in the GitHub distribution and explained in [23].

7.1 Comparing basis construction methods

Here we test and compare methods that construct a triangular or LLL-reduced basis, in up to 70 dimensions. We start with the basis $\boldsymbol{V}_0$ and apply LLL with $\delta = 0.5$ to obtain another basis $\boldsymbol{V}_1$ made of shorter vectors. Then we apply the lower-triangular construction method of Section 3 to $\boldsymbol{V}_1$ to obtain $\boldsymbol{V}_2$, we apply the upper-triangular method to $\boldsymbol{V}_1$ to obtain $\boldsymbol{V}_3$, and we apply it again to $\boldsymbol{V}_3$ to construct $\boldsymbol{V}_4$, just to see how much faster the algorithm runs when the basis is already upper-triangular. After that, we compute $\boldsymbol{V}_5$ as the m-dual of $\boldsymbol{V}_4$ as in Section 3, and we apply LLL with $\delta = 0.5$ to $\boldsymbol{V}_5$ to obtain a reduced m-dual basis $\boldsymbol{V}_6$. We transform $\boldsymbol{V}_6$ to a lower-triangular m-dual basis, then transform it to an upper-triangular m-dual basis, then repeat this with the old triangularization method of [5]. We repeat all of this for 1000 different multipliers a, in 10, 20, ..., 70 dimensions. For each type of transformation and each number of dimensions, we collect the total computing time for the 1000 multipliers. These times are reported in Figure 2.

We find that building a triangular primal basis or a lower-triangular m-dual basis is much faster than applying LLL, although computing an upper-triangular m-dual basis turns out to be slower than computing a lower-triangular one in this setting. The

Num. dimens.:	10	20	30	40	50	60	70
LLL 0.5	65522	218894	404905	664439	998318	1441736	1978508
LowTri	7506	20418	32563	45374	59742	75688	92373
UppTri	7504	20460	33221	45965	60206	75519	91994
UppTri2	3356	7445	12948	20297	29253	39084	50442
mDualUp	2336	11207	35725	82420	155578	256210	395274
LLLDual 0.5	39642	107380	148186	197315	258669	335396	431495
LowTriDual	26896	81010	115344	156796	206513	262508	325442
UppTriDual	24431	102341	226101	381063	568147	786156	1025855
UppTriDualOld	76137	369105	833980	1511613	2457780	3725551	5353556

Fig. 2. Timings for basis construction methods in microseconds, with $m = 1099511627791$.

triangularization algorithm runs faster when the basis is already triangular. The old triangulation algorithm (`UppTriDualOld`) is about five times slower than the new one.

We made a similar experiment to assess the speed and effectiveness of the LLL algorithm, using the same 1000 lattices. In the following, we use the notation LLLx to denote LLL with factor $\delta = 0.x$. From the initial basis $\boldsymbol{V}_0$, we first apply LLL5 ($\delta = 0.5$), then apply LLL8 to the resulting basis, then LLL99, and finally LLL99999 ($\delta = 0.99999$), each time recording the squared length of the shortest basis vector. After that, we apply LLL99999 directly to $\boldsymbol{V}_0$ (this is labeled LLL99999-new). We want to see if the incremental reduction could be faster and/or lead to a shorter shortest vector than the direct reduction with the large δ. Then, we transform the reduced basis to an upper-triangular one, compute its m-dual (which is lower triangular), and apply the same sequence of LLL reductions to this m-dual basis. We repeated this for the 1000 multipliers a.

Figure 3 reports the timings and the average square length of the shortest vector for each case, to assess the effectiveness of LLL. By comparing the average square lengths for LLL with $\delta = 0.5$ and $\delta = 0.99999$ in Figure 3, we see that taking δ closer to 1 gives significantly shorter vectors. Performing LLL by increasing δ incrementally is often 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 3, 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}. The behavior is similar for the m-dual.

7.2 Testing reduction methods and the BB

We now compare the performances of different pre-reduction strategies when finding a shortest nonzero vector with the BB algorithm, for both the primal and the m-dual, this

Timings for different methods, in microseconds (10^{-6} seconds):

Num. dimens.:	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

Average square length of shortest basis vectors:

Num. dimens.:	20	30	40	50	60
LLL5	2.1623e+23	9.0331e+23	1.2074e+24	1.2089e+24	1.2089e+24
LLL99999	1.1352e+23	4.2196e+23	9.4600e+23	1.2085e+24	1.2089e+24
LLL99999-pnew	1.1383e+23	4.2847e+23	9.7951e+23	1.2085e+24	1.2089e+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

Fig. 3. Timings in microseconds and average square length for LLL with $m = 1099511627791$.

time with 50 choices of multipliers. For each strategy, the basis is pre-reduced and then the BB algorithm is called to find a shortest vector. We compute the time to do this, the square length of the shortest basis vector, and the number of calls to the recursive BB procedure (the number of visited nodes in the BB tree). Figure 4 reports the average values for selected cases, for the m-dual lattices with the L^2 norm and using the Cholesky decomposition for the BB. Here, BKZx-k means BKZ with factor $\delta = 0.x$ and block size of k, and "+BB" means that BB was applied.

We find that applying BKZ with a large k before the BB is much more effective than applying LLL, especially in large dimensions, and using an incremental strategy for LLL does not bring much gain when BKZ is applied. In 20 dimensions or less, we do not see much difference between the pre-reduction methods applied before BB. But if we apply only LLL, the BB gets much slower than with BKZ, even with $\delta = 0.99999$, because the pre-reduction is not as good, so the BB tree has more nodes. If we apply only LLL with a smaller δ, such as LLL5 and LLL9, the BB fails frequently in the larger dimensions, because there are way too many nodes to examine in the BB tree. This is why LLL5+BB and LLL9+BB are not shown in the table. Doing the BB without any LLL or BKZ reduction is also totally impractical, it fails most of the time even in 5

```
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
BKZ99999-12+BB        517      2871     24310    152247  14199518
L5+L9+BKZ-12+BB       674      3355     24001    160298  12090483
Average square length of shortest basis vector:
LLL5              36783.7     255.0     46.82     40.64     36.38
LLL99999          36395.1     231.9     25.38     14.84     13.50
BKZ99999-10       36395.1     231.4     25.16     14.28     11.94
L5+L9+BKZ-10      36395.1     231.4     25.14     14.26     11.86
All +BB methods   36395.1     231.4     25.14     14.10     11.50
Average number of calls to the recursive BB procedure:
LLL99999+BB             5        16       793    129459  58161623
BKZ99999-12+BB          5        16       693     53608  11006050
L5+L9+BKZ-12+BB         5        16       668     56555   9309786
```

Fig. 4. Comparing pre-reductions for $m = 1099511627791$.

dimensions. Note that when BB is applied and succeeds, the length of the shortest basis vector must be the same for all pre-reduction methods.

For the first four rows of the table, only pre-reductions are applied (no "+BB"). These "methods" are much faster, but do not return a shortest vector, especially in large dimensions. For instance, 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!

Results for other values of m, higher dimensions, other `Real` types, for the primal and dual lattices, the L^1 and L^2 norms, bounds based on a triangular basis, etc., are given in [23]. We find there that computing a shortest vector with the L^1 norm is slower than with the L^2 norm with the Cholesky decomposition, and that using bounds based on a triangular basis is usually much slower than with the Cholesky decomposition.

7.3 Searching for a lattice with the best FOM

This example compares different methods for finding the 3 best LCG multipliers a among $100,000$ different ones, in terms of the FOM $M_{\boldsymbol{t}}^{(1)}$ for $\boldsymbol{t} = (t_1, \ldots, t_d) = (32, 32, 16, 12, 10)$, for either the primal or the m-dual lattices. *Method 1* computes all the terms of the FOM for each a (no early discarding) using BKZ+BB with $\delta = 0.99999$ and $k = 10$. *Method 2* applies the same BKZ+BB, but uses early discarding. *Method*

3 applies only LLL with $\delta = 0.99999$ and also use early discarding. *Method 4* uses two stages with early discarding at each stage. The first stage uses only LLL99999 and retains the 50 best multipliers, while the second stage tests the latter as in Method 3. *Method 5* does the same but replaces $\boldsymbol{t}$ by $\boldsymbol{t}_0 = (4, 32, 16, 12)$ on the first stage. Table 1 summarizes the results. It gives the CPU time (in seconds) for each method, for the primal and the m-dual. 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. Method 1 is much too slow and Method 4 is not really competitive with Methods 2, 3, 5. Methods 4 and 5 are also not guaranteed to return the three best a, because there is a chance that one of them could be eliminated in the first stage. In our experiment, all methods returned the same best, but Methods 4 and 5 sometimes missed the second and third best. Moreover, the two-stage approach was never faster in our tests.

Table 1. Timings for the search example with $m = 1099511627791$.

method	primal	m-dual
1. BKZ+BB, naive	2427.2	2253.2
2. BKZ+BB, discard	7.3	9.0
3. LLL only, discard	7.0	8.6
4. Two stages, stage 1 with LLL	13.4	15.1
stage 2 with BKZ+BB	0.024	0.03
5. Two stages, stage 1 with LLL and $\boldsymbol{t}_0$	7.2	9.6
stage 2 with BKZ+BB and $\boldsymbol{t}$	0.024	0.05

8 Conclusion

We have described Lattice Tester and showed examples of what it can do. Its main goal is to compute a shortest nonzero vector in an integral lattice. This is done via a BB integer programming algorithm. We have shown that applying strong pre-reductions such as LLL or BKZ before the BB is practically essential for the algorithm to be effective in large dimensions. LLL or BKZ alone often provide a very short vector by themselves, sometimes a shortest one, but not always, especially in large dimensions. The user guide [23] and the GitHub site [25] provide much more details about Lattice Tester, including algorithm descriptions, examples, a detailed description of the API, and the source code. We plan to use it in the near future to search for good parameters for new RNGs.

Acknowledgement

This work has been supported by the NSERC Discovery Grant RGPIN-2018-05795 to Pierre L'Ecuyer. Erwan Bourceret, Raymond Couture, Marc-Antoine Savard, Richard Simard, and Mamadou Thiongane contributed to the C++ code.

References

1. Afflerbach, L., Grothe, H.: Calculation of Minkowski-reduced lattice bases. Computing **35**, 269–276 (1985)
2. Bremner, R.M.: Lattice Basis Reduction: An Introduction to the LLL Algorithm and Its Applications. Pure and Applied Mathematics. Chapman & Hall, CRC Press (2012)
3. Conway, J.H., Sloane, N.J.A.: Sphere Packings, Lattices and Groups, 3rd edn. Grundlehren der Mathematischen Wissenschaften 290. Springer-Verlag, New York (1999)
4. Couture, R., L'Ecuyer, P.: Linear recurrences with carry as random number generators. In: Proceedings of the 1995 Winter Simulation Conference, pp. 263–267 (1995)
5. Couture, R., L'Ecuyer, P.: Orbits and lattices for linear random number generators with composite moduli. Mathematics of Computation **65**(213), 189–201 (1996)
6. Couture, R., L'Ecuyer, P.: Distribution properties of multiply-with-carry random number generators. Mathematics of Computation **66**(218), 591–607 (1997)
7. Coveyou, R.R., MacPherson, R.D.: Fourier analysis of uniform random number generators. Journal of the ACM **14**, 100–119 (1967)
8. Dick, J., Kritzer, P., Pillichshammer, F.: Lattice Rules: Numerical Integration, Approximation, and Discrepancy. Springer (2022)
9. Dieter, U.: How to calculate shortest vectors in a lattice. Mathematics of Computation **29**(131), 827–833 (1975)
10. Fincke, U., Pohst, M.: Improved methods for calculating vectors of short length in a lattice, including a complexity analysis. Mathematics of Computation **44**, 463–471 (1985)
11. Fishman, G.S.: Monte Carlo: Concepts, Algorithms, and Applications. Springer Series in Operations Research. Springer-Verlag, New York, NY (1996)
12. Helfrich, B.: Algorithms to construct Minkowski-reduced and Hermite-reduced lattice bases. Theoretical Computer Science **41**, 125–139 (1985)
13. Hickernell, F.J., Hong, H.S., L'Ecuyer, P., Lemieux, C.: Extensible lattice sequences for quasi-Monte Carlo quadrature. SIAM Journal on Scientific Computing **22**(3), 1117–1138 (2001)
14. Knuth, D.E.: The Art of Computer Programming, Volume 2: Seminumerical Algorithms, third edn. Addison-Wesley, Reading, MA (1998)
15. L'Ecuyer, P.: Random numbers for simulation. Communications of the ACM **33**(10), 85–97 (1990)
16. L'Ecuyer, P.: Combined multiple recursive random number generators. Operations Research **44**(5), 816–822 (1996)
17. L'Ecuyer, P.: Good parameters and implementations for combined multiple recursive random number generators. Operations Research **47**(1), 159–164 (1999)
18. L'Ecuyer, P.: Tables of linear congruential generators of different sizes and good lattice structure. Mathematics of Computation **68**(225), 249–260 (1999)
19. L'Ecuyer, P.: Uniform random number generation. In: S.G. Henderson, B.L. Nelson (eds.) Simulation, Handbooks in Operations Research and Management Science, pp. 55–81. Elsevier, Amsterdam, The Netherlands (2006). Chapter 3
20. L'Ecuyer, P.: Quasi-Monte Carlo methods with applications in finance. Finance and Stochastics **13**(3), 307–349 (2009)
21. L'Ecuyer, P.: Random number generation. In: J.E. Gentle, W. Haerdle, Y. Mori (eds.) Handbook of Computational Statistics, second edn., pp. 35–71. Springer-Verlag, Berlin (2012)
22. L'Ecuyer, P.: Randomized quasi-Monte Carlo: An introduction for practitioners. In: P.W. Glynn, A.B. Owen (eds.) Monte Carlo and Quasi-Monte Carlo Methods: MCQMC 2016, pp. 29–52. Springer, Berlin (2018)

23. L'Ecuyer, P.: Lattice Tester guide (2025). https://www-labs.iro.umontreal.ca/~lecuyer/papers.html
24. L'Ecuyer, P., Blouin, F., Couture, R.: A search for good multiple recursive random number generators. ACM Transactions on Modeling and Computer Simulation **3**(2), 87–98 (1993)
25. L'Ecuyer, P., Bourceret, E., Munger, D., Savard, M.A., Simard, R., Thiongane, M., Weiss, C.: Lattice Tester (2025). https://github.com/pierrelecuyer/latticetester
26. L'Ecuyer, P., Bourceret, E., Munger, D., Savard, M.A., Simard, R., Wambergue, P.: Latmrg (2022). https://github.com/umontreal-simul/LatMRG
27. L'Ecuyer, P., Couture, R.: An implementation of the lattice and spectral tests for multiple recursive linear random number generators. INFORMS Journal on Computing **9**(2), 206–217 (1997)
28. L'Ecuyer, P., Lemieux, C.: Variance reduction via lattice rules. Management Science **46**(9), 1214–1235 (2000)
29. L'Ecuyer, P., Marion, P., Godin, M., Puchhammer, F.: A tool for custom construction of QMC and RQMC point sets. In: A. Keller (ed.) Monte Carlo and Quasi-Monte Carlo Methods: MCQMC 2020, pp. 51–70. Springer, Berlin (2022)
30. L'Ecuyer, P., Munger, D.: On figures of merit for randomly-shifted lattice rules. In: H. Woźniakowski, L. Plaskota (eds.) Monte Carlo and Quasi-Monte Carlo Methods 2010, pp. 133–159. Springer-Verlag, Berlin (2012)
31. L'Ecuyer, P., Simard, R.: 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)
32. L'Ecuyer, P., Simard, R.: On the lattice structure of a special class of multiple recursive random number generators. INFORMS Journal on Computing **26**(2), 449–460 (2014)
33. L'Ecuyer, P., Touzin, R.: 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, pp. 683–689. IEEE Press (2000)
34. L'Ecuyer, P., Touzin, R.: On the Deng-Lin random number generators and related methods. Statistics and Computing **14**, 5–9 (2004)
35. L'Ecuyer, P., Wambergue, P., Bourceret, E.: Spectral analysis of the MIXMAX random number generators. INFORMS Journal on Computing **32**(1), 135–144 (2020)
36. Lemieux, C.: Monte Carlo and Quasi-Monte Carlo Sampling. Springer-Verlag (2009)
37. Lemieux, C., L'Ecuyer, P.: A comparison of Monte Carlo, lattice rules and other low-discrepancy point sets. In: H. Niederreiter, J. Spanier (eds.) Monte Carlo and Quasi-Monte Carlo Methods 1998, pp. 326–340. Springer-Verlag, Berlin (2000)
38. Lenstra, A.K., Lenstra, H.W., Lovász, L.: Factoring polynomials with rational coefficients. Math. Ann. **261**, 515–534 (1982)
39. Lovász, L.: An Algorithmic Theory of Numbers, Graphs and Convexity. No. 50 in SIAM CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, Philadelphia (1986)
40. Nguyen, P.Q., Vallée, B. (eds.): The LLL Algorithm: Survey and Applications. Springer Verlag, Berlin, Heidelberg (2010)
41. Pohst, M.: On the computation of lattice vectors of minimal length, successive minima and reduced bases with applications. ACM SIGSAM Bulletin **15**, 37–44 (1981)
42. Schnorr, C.P.: A hierarchy of polynomial time lattice basis reduction algorithms. Theoretical Computer Science **53**(2), 201–224 (1987)
43. Schnorr, C.P., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. In: L. Budach (ed.) Fundamentals of Computation Theory: 8th International Conference, pp. 68–85. Springer-Verlag, Berlin, Heidelberg (1991)
44. Shoup, V.: NTL: A Library for doing Number Theory. Courant Institute, New York University, New York, NY (2018). https://shoup.net/ntl/
45. Sloan, I.H., Joe, S.: Lattice Methods for Multiple Integration. Clarendon Press, Oxford (1994)