Les Cahiers du GERAD ISSN: 0711–2440

A general software tool for constructing rank-1 lattice rules

P. L'Ecuyer
D. Munger

G–2015–28

April 2015

GERAD HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

A general software tool for constructing rank-1 lattice rules

Pierre L'Ecuyer [a,b]

David Munger [a]

[a] *Département d'informatique et de recherche opérationnelle, Université de Montréal, Montréal (Québec) Canada, H3C 3J7*

[b] *GERAD, HEC Montréal, Montréal (Québec) Canada, H3T 2A7*

lecuyer@iro.umontreal.ca
david.munger@umontreal.ca

April 2015

Les Cahiers du GERAD

G–2015–28

Abstract: We introduce a new software tool and library named Lattice Builder, written in C++, that implements a variety of construction algorithms for good rank-1 lattice rules. It supports exhaustive and random searches, as well as component-by-component (CBC) and random CBC constructions, for any number of points, and for various measures of (non)uniformity of the points. The measures currently implemented are all shift-invariant and represent the worst-case integration error for certain classes of integrands. They include for example the weighted $\mathcal{P}_\alpha$ square discrepancy, the $\mathcal{R}_\alpha$ criterion, and figures of merit based on the spectral test, with projection-dependent weights. Each of these measures can be computed as a finite sum. For the $\mathcal{P}_\alpha$ and $\mathcal{R}_\alpha$ criteria, efficient specializations of the CBC algorithm are provided for projection-dependent, order-dependent and product weights. For numbers of points that are integer powers of a prime base, the construction of embedded rank-1 lattice rules is supported through any of the above algorithms, and also through a fast CBC algorithm, with a variety of possibilities for the normalization of the merit values of individual embedded levels and for their combination into a single merit value. The library is extensible, thanks to the decomposition of the algorithms into decoupled components, which makes it easy to implement new types of weights, new search domains, new figures of merit, etc.

Key Words: Lattice rules, figures of merit, quasi-Monte Carlo, multidimensional integration, CBC construction.

Acknowledgments: This work has been supported by NSERC-Canada grant No. ODGP0110050 and a Canada Research Chair to the first author. Computations were performed using the infrastructure from the Réseau québécois de calcul haute performance (RQCHP), a member of the Compute Canada network. We are very grateful to Dirk Nuyens for testing the software and for his several comments and corrections on both the software and the article. Mohamed Hanini also helped testing the software and improve its user's guide. We thank the Associate Editor Ronald Cools and the anonymous referees for their help in improving the paper.

1 Introduction

Lattice rules are often used as a replacement for Monte Carlo (MC) to integrate multidimensional functions. To estimate the integral, say over the unit hypercube of volume one in s dimensions, $[0,1)^s$, the simple MC method samples the integrand at n independent random points having the uniform distribution in the hypercube, and takes the average. These independent random points tend to spread irregularly, with clusters and gaps, over the integration region (the unit hypercube). Quasi-Monte Carlo (QMC) methods, which include lattice rules, aim at sampling at a set of (structured) points that cover the integration region more evenly than MC, i.e., with a low discrepancy with respect to the uniform distribution. With a lattice rule, these points are the points of an integration lattice that fall in the unit hypercube (see the next section). In randomized QMC (RQMC), the structured points are randomized in a way that the point set keeps its good uniformity, while each individual point has a uniform distribution over the integration region, so the average is an unbiased estimator of the integral. This randomization can be replicated independently if one wishes to estimate the variance. When the integrand is smooth enough, QMC (resp., RQMC) can reduce the integration error (resp., the variance of the estimator) significantly compared with MC, with the same number n of function evaluations. For detailed background on QMC, RQMC, and lattice rules, see Niederreiter [1992a], Sloan and Joe [1994], L'Ecuyer and Lemieux [2000], L'Ecuyer [2009], Lemieux [2009], Dick and Pillichshammer [2010], Nuyens [2014], and the references given there.

A lattice rule is defined by a set of numerical parameters that determine its point set. These lattice parameters are selected to (try to) minimize a measure of non-uniformity of the points, which should depend in general on the class of integrands that we want to consider. The non-uniformity measure is also parameterized (typically) by continuous parameters (e.g., weights given to the quality of the lower-dimensional projections of the point set to suit specific classes of integrand, which yields a *weighted figure of merit*), so there is an infinite number of possibilities. The lattice parameters would also depend on the desired type of lattice, dimension, and number of points. It is clearly impossible to search and tabulate once and for all the best lattice rules for all these possibilities. We need a tool that can construct good integration lattices on demand, with an arbitrary number of points, in an arbitrary dimension, for various ways of measuring the uniformity of the points (so one can used a figure of merit adapted to the problem at hand), and with various construction methods. The lack of such a tool so far has hindered the widespread use of lattice rules in simulation experiments. Other methods such as Sobol' point sets and sequences turn out to be more widely used even though (randomly-shifted) lattice rules are easier to implement, because robust general purpose parameters are more easily available for the former. The main purpose of the *Lattice Builder* software tool proposed here is to fill this gap. This tool is also very handy for doing research on lattice rules and we give a few illustrations of that in the paper. When searching for good lattice rules for a particular application, the CPU time required to search for a good rule is usually much smaller than the CPU time for running the experiments using the rule. We stress that the search does not have to be exhaustive among all possibilities; typically, a rule that is good enough can be found very quickly with Lattice Builder after examining only a tiny fraction of the possibilities.

Integration lattices and other QMC or RQMC point sets are also used for other applications such as function approximation, solving stochastic partial differential equations, global optimization of a function, etc.; see, e.g., Niederreiter [1992b], Kuo et al. [2008], L'Ecuyer and Owen [2009] and Woźniakowski and Plaskota [2012]. Lattice Builder can find lattices not only for multivariate integration, but for these other applications as well, with an appropriate choice for the measure of non-uniformity (or discrepancy from the uniform distribution) of the points.

Lattice Builder is implemented in the form of a C++ software library, which can be used from other programs via its application programming interface (API). It also offers an executable program that can be called either via a command-line interface (CLI) or via a graphical web interface to search for a good lattice with specified constraints and criteria. It is available for download from the first author's web page and other distribution sites.

The remainder of the paper is organized as follows. In Section 2, we recall some background on RQMC methods and lattice rules, discuss previous related work, existing software and tables, and summarize the

main features of Lattice Builder, including software engineering techniques used to speed up the execution. In Sections 3 and 4, we review the theoretical objects and algorithms implemented by the software tool and we explain how we generalized certain methods, for example by adapting existing search algorithms to support more general parameterizations of figures of merit, or to improve on certain computational aspects required in the implementation. A primary goal was to make the search for good lattices as fast as possible while allowing flexibility in the choice of figures of merits and search algorithms. Section 5 gives a few comments on the interfaces and usage of Lattice Builder. In Section 6, we provide examples of applications where we compare the behavior and distribution of different figures of merit and different choices of the weights.

2 Lattice rules and the purpose of lattice builder

2.1 MC, QMC, and RQMC integration

The MC method is widely used to estimate the expectation $\mu = \mathbb{E}[X]$ of a real-valued random variate X by computing the arithmetic average of n independent simulated realizations of X. In practice, X is simulated by generating, say, s (pseudo)random numbers uniformly distributed over $(0,1)$ and transforming them appropriately. That is, we can write $X = f(\mathbf{U})$ for some function $f : \mathbb{R}^s \to \mathbb{R}$, where $\mathbf{U}$ is a random vector uniformly distributed in $(0,1)^s$, and

$$\mu = \mathbb{E}[X] = \mathbb{E}[f(\mathbf{U})] = \int_{[0,1)^s} f(\mathbf{u})\,\mathrm{d}\mathbf{u}.$$

(If s is random, we can take an upper bound or even ∞ in the above expression.) The crude MC estimator of μ averages n independent replications of $f(\mathbf{U})$:

$$\hat{\mu}_{n,\mathrm{MC}} = \frac{1}{n}\sum_{i=0}^{n-1} f(\mathbf{U}_i), \tag{1}$$

where $\mathbf{U}_0, \ldots, \mathbf{U}_{n-1}$ are n independent points uniformly distributed in $(0,1)^s$. MC integration is easy to implement but its variance $\mathrm{Var}[\hat{\mu}_{n,\mathrm{MC}}] = \mathbb{E}[(\hat{\mu}_{n,\mathrm{MC}} - \mu)^2] = \mathcal{O}(n^{-1})$ converges slowly as a function of n.

The idea behind QMC is to replace the n independent random points by a set P_n of n points that cover the unit hypercube $[0,1)^s$ more evenly, to reduce the integration error [Niederreiter, 1992b; Sloan and Joe, 1994; Dick and Pillichshammer, 2010]. In ordinary QMC, these points are deterministic. With RQMC, the n points $\mathbf{U}_0, \ldots, \mathbf{U}_{n-1}$ in (1) are constructed in a way that $\mathbf{U}_i$ is uniformly distributed over $[0,1)^s$ for each i, but in contrast with MC, these points are not independent and are constructed so that they keep their QMC property of covering the unit hypercube very evenly. The first property implies that the average is an unbiased estimator of μ, while its variance can be smaller than for MC because of the second property. Under appropriate conditions on the integrand, the variance can be proved to converge at a faster rate than for MC, as a function of n. Overviews of prominent RQMC methods can be found in L'Ecuyer [2009] and Lemieux [2009]. Here we shall focus on randomly-shifted lattice rules, discussed in Section 2.2 below.

2.2 Rank-1 lattice rules

An *integration lattice* is a vector space of the form

$$L_s = \left\{\mathbf{v} = \sum_{j=1}^{s} h_j \mathbf{v}_j \text{ such that } h_j \in \mathbb{Z} \text{ for all } j = 1, \ldots, s\right\},$$

where $\mathbf{v}_1, \ldots, \mathbf{v}_s \in \mathbb{R}^s$ are linearly independent over $\mathbb{R}$ and where L_s contains the set of integer vectors $\mathbb{Z}^s$. This implies that all the components of each $\mathbf{v}_j$ are multiples of $1/n$, where n is the number of lattice points per unit of volume. A *lattice rule* is the QMC method obtained by replacing the n independent uniform points $\mathbf{U}_0, \ldots, \mathbf{U}_{n-1}$ in (1) by the point set $P_n = \{\mathbf{u}_0, \ldots, \mathbf{u}_{n-1}\} = L_s \cap [0,1)^s$ [Sloan and Joe, 1994].

A randomized counterpart of P_n can be obtained by applying the same *random shift* $\mathbf{U}$ uniformly distributed in $[0,1)^s$, modulo 1, to each point of P_n:

$$P'_n = \{\mathbf{U}_i = (\mathbf{u}_i + \mathbf{U}) \bmod 1 \,:\, i = 0, \ldots, n-1\}, \tag{2}$$

where the modulo 1 applies componentwise. The RQMC estimator $\hat{\mu}_{n,\mathrm{RQMC}}$ obtained with the points $\mathbf{U}_0, \ldots, \mathbf{U}_{n-1}$ of P'_n is called a *randomly-shifted lattice rule* [L'Ecuyer and Lemieux, 2000]. It was proposed by Cranley and Patterson [1976].

The *rank* of L_s is the smallest r such that one can find a basis of the form $\mathbf{v}_1, \ldots, \mathbf{v}_r, \mathbf{e}_{r+1}, \cdots, \mathbf{e}_s$, where $\mathbf{e}_j$ is the j-th unit vector in s-dimensions. For a lattice of rank 1, the point set P_n can be written as

$$P_n = \{\mathbf{u}_i = (i\mathbf{a}/n) \bmod 1 = i\mathbf{v}_1 \bmod 1 \,:\, i = 0, \ldots, n-1\}, \tag{3}$$

where $\mathbf{a} = (a_1, \ldots, a_s) \in \mathbb{Z}^s$ is an integer *generating vector* and $\mathbf{v}_1 = \mathbf{a}/n$. We call this P_n a *rank-1 lattice point set*. A *Korobov* rule is a lattice rule of rank 1 whose generating vector has the special form $\mathbf{a} = (1,\ a,\ a^2 \bmod n,\ \ldots,\ a^{s-1} \bmod n)$ for some $a \in \mathbb{Z}_n^* = \{1, \ldots, n-1\}$.

Each projection of an integration lattice L_s over a subset of coordinates $\mathfrak{u} \subseteq \{1, \ldots, s\}$ is also an integration lattice $L_s(\mathfrak{u})$ whose corresponding point set $P_n(\mathfrak{u})$ is the projection of P_n over the coordinates in $\mathfrak{u}$. It is customary to write the QMC integration error (and the RQMC variance) for f as sums of errors (and variances) for integrating some projections $f_\mathfrak{u}$ of f by the projected points $P_n(\mathfrak{u})$, over all subsets $\mathfrak{u} \subseteq \{1, \ldots, s\}$, and to reduce each term of the sum we want each projection $P_n(\mathfrak{u})$ to cover the unit cube $[0,1)^{|\mathfrak{u}|}$ as evenly as possible [L'Ecuyer, 2009]. In particular, it is preferable that all the points of each projection be distinct. L'Ecuyer and Lemieux [2000] call a lattice rule *fully projection-regular* if $P_n(\mathfrak{u})$ contains n distinct points for all $\mathfrak{u} \neq \varnothing$, i.e., if the points never superpose on each other in lower-dimensional projections. This happens if and only if the rule has rank 1 and the coordinates of $\mathbf{a}$ are all relatively prime with n [L'Ecuyer and Lemieux, 2000].

Currently, Lattice Builder considers only fully projection-regular rules, which must be of rank 1. In the rest of this paper, for simplification, the term *lattice rule* will always refer to fully projection-regular rank-1 rules. All the figures of merit that we consider are shift-invariant, i.e. their values do not change with the random shift, so we express them in terms of the deterministic point set P_n directly. Further coverage of randomly-shifted lattice rules can be found in L'Ecuyer and Lemieux [2000], Kuo et al. [2006], L'Ecuyer et al. [2010], L'Ecuyer and Munger [2012], and the references given there.

Figure 1 illustrates the point sets P_n for two rank-1 lattices with $n = 100$, one with $\mathbf{a} = (1, 23)$ and the other with $\mathbf{a} = (1, 3)$, in the upper panels. Clearly, the first point set has better uniformity than the second. Applying a random shift will slide all the points together (modulo 1), keeping their general structure intact. For comparison, the lower left panel shows a centered rectangular grid with $n = 100$ points, whose projection to the first coordinate contains only 10 distinct points, and similarly for the second coordinate. This is actually the point set of a rank-2 lattice rule with $n = 100$, $\mathbf{v}_1 = (1/10, 0)$, and $\mathbf{v}_2 = (0, 1/10)$, shifted by the vector $\mathbf{U} = (1/20, 1/20)$. In the lower right panel, we have 100 random points, which cover the space much less evenly than the lattice points in the upper left.

As illustrated in the figure, the choice of $\mathbf{a}$ can make a significant difference in the uniformity of P_n. Another obvious example of a bad choice is when all coordinates of $\mathbf{a}$ are equal; then all the points of P_n are on the diagonal line from $(0, \ldots, 0)$ to $(1, \ldots, 1)$. To measure the quality of lattice rules and find good values of $\mathbf{a}$, Lattice Builder looks at the uniformity of the projections $P_n(\mathfrak{u})$ for $\mathfrak{u} \subseteq \{1, \ldots, s\}$ (either all of them or a subset), and tries to minimize a *figure of merit* that combines some measures of the non-uniformity of the projections $P_n(\mathfrak{u})$. This process is implemented for various figures of merit and search spaces for $\mathbf{a}$, as explained in Sections 3 and 4.

2.3 Embedded lattice rules

QMC or RQMC estimators sometimes need to be refined by increasing the number of points, and most preferably without throwing away the previous function evaluations. This can be achieved by using embedded

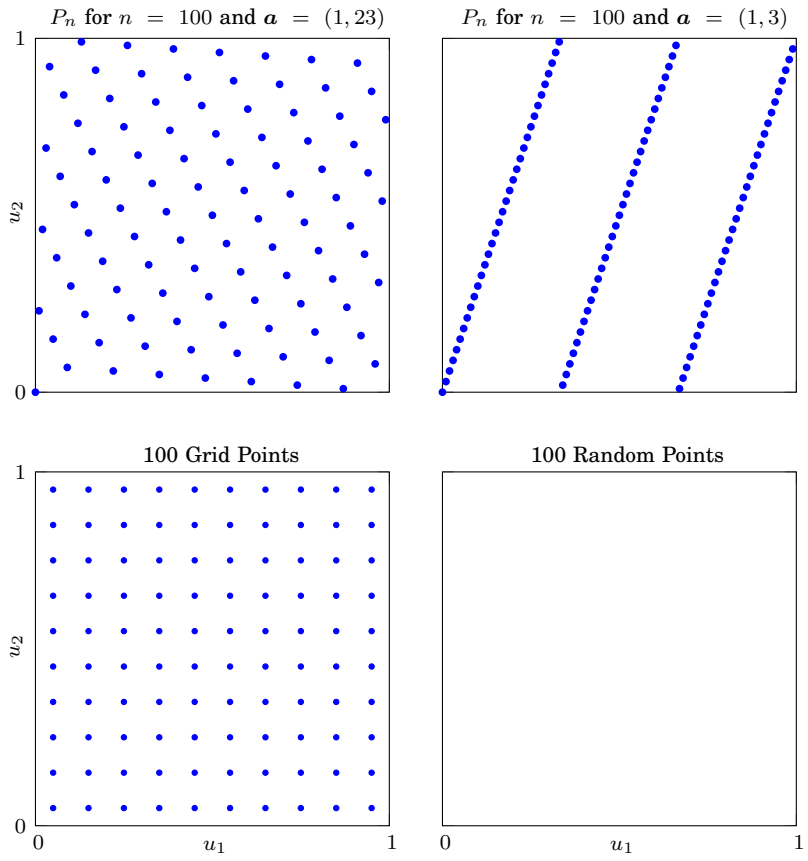

Figure 1: Comparison of two-dimensional point sets in the unit square: lattice points P_n with $n = 100$ and $\mathbf{a} = (1, 23)$ (top left) and $\mathbf{a} = (1, 3)$ (top right), 10×10 grid points (bottom left), and 100 random points (bottom right).

point sets $P_{n_1} \subset P_{n_2} \subset \ldots P_{n_m}$ with increasing numbers of points $n_1 < n_2 < \cdots < n_m$, for some positive integer m (the maximum number of nested levels). With lattice rules, this means taking $n_k = b^k$ for each embedded level k and for some prime base b, while keeping the same generating vector $\mathbf{a}$ for all embedded levels (or equivalently $\mathbf{a}_k = \mathbf{a}_{k+1} \bmod n_k$ at level k for all $k < m$), and the same random shift. Such RQMC estimators are called *embedded lattice rules* [Hickernell et al., 2001; Cools et al., 2006]. When $m = \infty$, they are called *extensible* [Dick et al., 2008]. For practical reasons, Lattice Builder assumes $m < \infty$. Choosing a good generating vector $\mathbf{a}$ for embedded lattice rules requires a figure of merit that reflects the quality of the different embedded levels. This is dealt with in Section 3.5 below. Lattice Builder also supports the extension of the number of points by modifying the generating vector of a given sequence of embedded lattice rules so that m can be incremented without affecting the point sets at levels up to its current value.

The points in successive embedded levels can be enumerated elegantly as a *lattice sequence* by using a permutation based on the radical inverse function [Hickernell et al., 2001]. Or, more simply, the new points added from level $k-1$ to level k are given by

$$P_{b^k} \setminus P_{b^{k-1}} = \left\{ \mathbf{U}_{(ib+j)b^{m-k}} \,:\, i = 0, \ldots, b^{k-1} - 1;\; j = 1, \ldots, b-1 \right\},$$

for $k = 1, \ldots, m$, starting with $P_{b^0} = \{\mathbf{U}_0\}$, and where $\mathbf{U}_j$ is the j-th point from the point set P_{n_m} at the highest level, given by (2) with $n = b^m$. We shall refer to the lattices described in Section 2.2 as *ordinary lattices* when we need to differentiate them from embedded lattices.

2.4 Previous work and existing tools

Currently available software tools for finding good parameters for lattice rules apply to a limited selection of figures of merit and of algorithms. To our knowledge, there is none at the level of generality of Lattice Builder. Nuyens [2012] provides Matlab code for the fast component-by-component (CBC) construction of ordinary and embedded lattice rules with product and order-dependent weights. Precomputed tables of good parameters for lattice rules can also be found in published articles, books, and websites; see, for example, Maisonneuve [1972], Sloan and Joe [1994], L'Ecuyer and Lemieux [2000], and Kuo [2012]. These parameters were found by making certain assumptions on the integrand, which are not necessarily appropriate in specific practical applications. The main drawback of such tables is that it is not possible to tabulate good lattice parameters in advance for all numbers of points, all dimensions, and any type of figure of merit that one may need.

Software is also available for *using* lattice rules in RQMC settings. For example, the Java simulation library SSJ [L'Ecuyer, 2008] permit one to replace easily any stream of uniform random numbers by QMC or RQMC points, including those of an arbitrary lattice rule. Burkardt [2012] offers C++, Fortran, and Matlab code for several QMC methods, including lattice rules. Lemieux [2012] provides C code for using lattice rules of the Korobov type, as part of a library that supports QMC methods.

2.5 Features of lattice builder

Lattice Builder permits one to find good lattice parameters for figures of merit that give arbitrary weights to the projections $P_n(\mathfrak{u})$, so the weights can be tailored to a given problem. It can be used as a standalone tool or invoked from another program to construct an integration lattice when needed, with the appropriate dimension, number of points, and figure of merit that may not be known in advance. It allows researchers to study empirically the properties of various figures of merit such as the distribution of values of a figure of merit over a given family of lattices, or the joint distribution for two or more figures of merit, etc. It can also be used to compare the behavior and properties of different search algorithms, or simply to evaluate the quality of a given lattice rule, or even search for bad lattices.

Cools et al. [2006] opened the way to efficient search algorithms with their fast CBC algorithm that reuses intermediate results during the computation of a figure of merit, for special parameterizations of the weights given to the projections. We extend these methods to more general weight parameterizations. Lattice Builder supports various combinations of types of figures of merit and search methods not found in existing software, and offers enough flexibility to easily add new such combinations in the future.

Such generality and flexibility requires decomposing the search process into decoupled software components, each corresponding to distinct tasks that are part of the search process (e.g., enumerating candidate generating vectors) and that can be performed using different approaches (e.g., enumerating all possible vectors or choosing a few at random). The software offers a choice of alternative implementations for each of these tasks. The usual approach consists of defining for each task an application programming interface (API), i.e., a set of functionalities and properties that specifies precisely how a user of the API can communicate with a software component that performs a given task (e.g., how to get the next candidate vector), without telling how it is implemented. Thus, various objects can implement the same interface in distinct ways. This is called *polymorphism*. The traditional object-oriented approach relies on *dynamic polymorphism*, that is, when a user calls a function from a given interface, the particular implementation that must be used is resolved at runtime every time the function is called. This consumes CPU time and can cause significant overhead if this resolution process takes a time comparable to that required to actually execute the function. This is especially true for a function that does very little work, like mapping indices of a symmetric vector, and that is called relatively frequently, as in a loop that is repeated a large number of times, which is common in our software. Dynamic polymorphism thus prevents the compiler from performing certain kinds of optimizations, such as *function inlining*. We managed to retain good performance by designing the code in a way that the compiler itself can resolve the polymorphic functions. We did this through the use of C++ templates that act as a code generation tool. This is known as *static polymorphism* [Alexandrescu, 2001;

Lischner, 2009]. In addition to moving the resolution process from runtime to compile time, this approach allows the compiler to perform further optimizations.

Of course, using C++ libraries directly from other languages is not easy in general. However, the `latbuilder` *executable* program should be able to carry the most common tasks for a majority of users, and it is reasonably simple to call this executable from other languages such as C, Java, and Python, for example. Contributors are also welcome to write interface layers for the library in other languages.

Designing these decoupled, heterogeneous components was not straightforward. For example, the fast CBC construction method described in Section 4.4 evaluates a figure of merit for all candidate lattice rules simultaneously, in contrast to other construction methods which normally evaluate the figure of merit for one lattice at a time, so accessing the values of a figure of merit for the different lattice rules sequentially must be implemented differently. To make this transparent to the user at the API level, we use an iterator design closely inspired from that of the containers in the C++ standard template library (STL), which relies on code generation through class templates rather than on polymorphism.

3 Figures of merit

In this section, we describe the different types of figures of merit currently implemented in Lattice Builder. Our generic notation for a figure of merit is $\mathcal{D}_q(P_n)$. In the QMC literature, this is a standard notation for discrepancies, which measure the distance (in some sense) between the distribution of the points of P_n and the uniform distribution over $[0,1)^s$. Here, we broaden its usage to positive real-valued figures of merit that are not necessarily discrepancies. Our figures of merit are weighted combinations of measures of non-uniformity of the projections $P_n(\mathfrak{u})$, as we now explain.

3.1 Weighted figures of merit

3.1.1 The ANOVA decomposition and weighted projections

The general figures of merit supported in Lattice Builder are expressed as a weighted ℓ_q-norm with respect to the projections $P_n(\mathfrak{u})$ of P_n:

$$[\mathcal{D}_q(P_n)]^q = \sum_{\varnothing \neq \mathfrak{u} \subseteq \{1,\dots,s\}} \gamma_{\mathfrak{u}}^q \, [\mathcal{D}_{\mathfrak{u}}(P_n)]^q \,, \tag{4}$$

where $q \geq 1$ is a real number (the most common choice is $q = 2$) and where for every set of coordinates $\mathfrak{u}$, the *projection-dependent weight* $\gamma_{\mathfrak{u}}^q$ is a real-valued (finite) constant and the *projection-dependent figure of merit* $\mathcal{D}_{\mathfrak{u}}(P_n) = \mathcal{D}_{\mathfrak{u}}(P_n(\mathfrak{u}))$ depends only on $P_n(\mathfrak{u})$. The weights $\gamma_{\mathfrak{u}}^q$ can be set to any real numbers, and there are several choices for $\mathcal{D}_{\mathfrak{u}}$, described further in this section. When searching for good lattices, all weights $\gamma_{\mathfrak{u}}^q$ should be non-negative, but our software can handle negative values of $\gamma_{\mathfrak{u}}^q$ as well. This could be useful for experimental purposes, e.g., if one seeks a lattice that is good in most projections but bad in one or more particular projection(s), or if we want to add a negative correction to the weight of some projection when combining different types of weights. Here we refer to $\gamma_{\mathfrak{u}}^q$ instead of $\gamma_{\mathfrak{u}}$ for the weights to allow for negative weights. Of course, error and variance bounds based on Hölder's inequality, as in (11) for example, are not valid with negative weights. Other authors sometimes use a different formulation of the ℓ_q norm, for example Nuyens [2014] who takes the ℓ_q norm with respect to the terms of the Fourier expansion (the sum is over $\mathbb{Z}^s$). We also note that the presence of q in (11) does not really enlarge the class of figures of merit that can be considered, because one can always redefine $\gamma_{\mathfrak{u}}$ and $\mathcal{D}_{\mathfrak{u}}(P_n)$ in a way that they incorporate the power q. But this parameter q can be convenient for proving integration error or variance bounds for various classes of functions via Hölder's inequality.

The general figure of merit (4) is related to (and motivated by) the functional ANOVA decomposition of f [Efron and Stein, 1981; Owen, 1998; Sobol', 2001; Liu and Owen, 2006]:

$$f(\mathbf{x}) = \mu + \sum_{\varnothing \neq \mathfrak{u} \subseteq \{1,\dots,s\}} f_{\mathfrak{u}}(\mathbf{x}),$$

where, for each non-empty $\mathfrak{u} \subseteq \{1, \ldots, s\}$, the ANOVA component $f_{\mathfrak{u}}(\mathbf{x})$ integrates to zero and depends only on the components of $\mathbf{x}$ whose indices are in $\mathfrak{u}$ (its definition can be found in the above references), these components are orthogonal with respect to the $\mathcal{L}_2$ inner product, and the variance of the RQMC estimator decomposes in the same way:

$$\mathrm{Var}[\hat{\mu}_{n,\mathrm{RQMC}}] = \sum_{\varnothing \neq \mathfrak{u} \subseteq \{1,\ldots,s\}} \mathrm{Var}[\hat{\mu}_{n,\mathrm{RQMC},\mathfrak{u}}], \tag{5}$$

where $\hat{\mu}_{n,\mathrm{RQMC},\mathfrak{u}}$ is the RQMC estimator for $\int_{[0,1)^s} f_{\mathfrak{u}}(\mathbf{x})\, d\mathbf{x}$ using the same points as $\hat{\mu}_{n,\mathrm{RQMC}}$. Thus, the RQMC variance for f is the sum of RQMC variances for the ANOVA components $f_{\mathfrak{u}}$. Minimizing this RQMC variance would be equivalent to minimizing (4) if $\gamma_{\mathfrak{u}}^q\, [\mathcal{D}_{\mathfrak{u}}(P_n)]^q$ was exactly proportional to $\mathrm{Var}[\hat{\mu}_{n,\mathrm{RQMC},\mathfrak{u}}]$, for all $\mathfrak{u}$, and $q < \infty$. In the former expression, $\mathcal{D}_{\mathfrak{u}}(P_n)$ pertains to the quality of the projection $P_n(\mathfrak{u})$, while the weight $\gamma_{\mathfrak{u}}^q$ should reflect the variability of $f_{\mathfrak{u}}$ or, more precisely, its anticipated contribution to the variance (more on this later; see also L'Ecuyer and Munger 2012). Note that all *one-dimensional projections* (before random shift) are the same. So the weights $\gamma_{\mathfrak{u}}^q$ for $|\mathfrak{u}| = 1$ are irrelevant, if we assume that all computations are exact. In practice, however, computations are in floating point (they are not exact) and these weights may have an impact on what vector $\mathbf{a}$ is retained.

For $q = \infty$, the sum in (4) is interpreted by Lattice Builder as a maximum that retains only the worst weighted value:

$$\mathcal{D}_\infty(P_n) = \max_{\varnothing \neq \mathfrak{u} \subseteq \{1,\ldots,s\}} \gamma_{\mathfrak{u}}\, \mathcal{D}_{\mathfrak{u}}(P_n), \tag{6}$$

and the user specifies the weights as $\gamma_{\mathfrak{u}}$ rather than $\gamma_{\mathfrak{u}}^q$. The set operators $\sum$ (sum) and max (maximum) in the context of (4) and (6) are implemented with distinct algorithms, and we refer to them as *accumulators* henceforth.

3.1.2 Saving work

In some situations, $\mathcal{D}_q$ must be evaluated for a j-dimensional lattice with generating vector $(a_1, \ldots, a_j)$ when the merit value for the $(j-1)$-dimensional lattice with generating vector $(a_1, \ldots, a_{j-1})$ is already available. In that case, Lattice Builder reuses the work already done, as follows. If $\mathcal{D}_{q,j}(P_n)$ denotes the contribution to (4) that depends only on the first j coordinates of P_n, we can write the recurrence

$$[\mathcal{D}_{q,j}(P_n)]^q = \sum_{\varnothing \neq \mathfrak{u} \subseteq \{1,\ldots,j\}} [\gamma_{\mathfrak{u}}\, \mathcal{D}_{\mathfrak{u}}(P_n)]^q = [\mathcal{D}_{q,j-1}(P_n)]^q + \sum_{\mathfrak{u} \subseteq \{1,\ldots,j-1\}} \left[\gamma_{\mathfrak{u}\cup\{j\}}\, \mathcal{D}_{\mathfrak{u}\cup\{j\}}(P_n)\right]^q, \tag{7}$$

for $j = 1, \ldots, s$, with $\mathcal{D}_{q,0}(P_n) = 0$. In particular, $\mathcal{D}_{q,s}(P_n) = \mathcal{D}_q(P_n)$. Lattice Builder stores $\mathcal{D}_{q,j-1}(P_n)$ to accelerate the computation of $\mathcal{D}_{q,j}(P_n)$. The time complexity to evaluate (7) depends on the time complexity to evaluate each $\mathcal{D}_{\mathfrak{u}}(P_n)$, but in the general cases it requires $2^j - 1$ such evaluations. We will see later that considerable savings are possible for certain choices of discrepancy and weight structure.

It is also possible in Lattice Builder to prevent the complete term-by-term evaluation of the sum in (4) by specifying an *early exit condition* on the value of the partial sum, which is checked after adding each new term. This can be used to reject a candidate P_n as soon as the sum is known to exceed some threshold, e.g., the merit value of the best candidate P_n found by the construction algorithm so far.

Lattice Builder implements specific types of projection-dependent figures of merit, described below, and can be easily extended to implement other projection-dependent figures of merit.

3.1.3 The $\mathcal{P}_\alpha$ criterion

One common figure of merit supported by Lattice Builder is based on the $\mathcal{P}_\alpha$ square discrepancy (see Sloan and Joe 1994; Hickernell 1998a; Nuyens 2014 and the references given there):

$$\mathcal{D}_{\mathfrak{u}}^2(P_n) = \frac{1}{n} \sum_{i=0}^{n-1} \prod_{j \in \mathfrak{u}} p_\alpha((i a_j/n) \bmod 1), \tag{8}$$

where

$$p_\alpha(x) = \frac{-(-4\pi^2)^{\alpha/2} B_\alpha(x)}{\alpha!} \tag{9}$$

with $\alpha = 2, 4, \ldots$, and B_α is the Bernoulli polynomial of even degree α. The $\mathcal{P}_\alpha$ criterion is actually defined for any $\alpha > 1$ [Sloan and Joe, 1994; Hickernell, 1998a], but its expression as a finite sum given in (8) holds only for $\alpha = 2, 4, \ldots$. With $\mathcal{D}_{\mathfrak{u}}^2$ as defined in (8), $\mathcal{D}_2^2$ is the weighted $\mathcal{P}_\alpha$ square discrepancy [Dick et al., 2004, 2006] and it is known (see also L'Ecuyer and Munger 2012, Section 4) that for all $n \geq 3$ and even α,

$$\mathrm{Var}[\hat{\mu}_{n,\mathrm{RQMC},\mathfrak{u}}] \leq \mathcal{V}_{\mathfrak{u}}^2(f)\, \mathcal{D}_{\mathfrak{u}}^2(P_n), \tag{10}$$

where

$$\mathcal{V}_{\mathfrak{u}}^2(f) = (2\pi)^{-\alpha|\mathfrak{u}|} \int_{[0,1)^{|\mathfrak{u}|}} \left| \frac{\partial^{\alpha|\mathfrak{u}|/2} f_{\mathfrak{u}}(\mathbf{x})}{\partial \mathbf{x}_{\mathfrak{u}}^{\alpha/2}} \right|^2 d\mathbf{x},$$

is the square variation of $f_{\mathfrak{u}}$ [Hickernell, 1998a], where $\partial^{\alpha|\mathfrak{u}|/2} f_{\mathfrak{u}}(\mathbf{x}) / \partial \mathbf{x}_{\mathfrak{u}}^{\alpha/2}$ denotes the mixed partial derivative of order $\alpha/2$ of $f_{\mathfrak{u}}$ with respect to each coordinate in $\mathfrak{u}$. This $\mathcal{V}_{\mathfrak{u}}^2(f)$ measures the variability of $f_{\mathfrak{u}}$. If $\mathcal{V}_{\mathfrak{u}}^2(f) < \infty$ for each $\mathfrak{u}$ and we take $\gamma_{\mathfrak{u}}^2 = \mathcal{V}_{\mathfrak{u}}^2(f)$, so the weights correspond to the variabilities of the projections, by combining (10), (5), and (4) for $q = 2$, we obtain that

$$\mathrm{Var}[\hat{\mu}_{n,\mathrm{RQMC}}] \leq \mathcal{D}_2^2(P_n). \tag{11}$$

In fact, the variance bound (11) holds for all integrands f for which

$$\mathcal{V}_{\mathfrak{u}}^2(f) \leq \gamma_{\mathfrak{u}}^2 < \infty. \tag{12}$$

The worst-case function f that satisfies Condition (12), and for which the RQMC variance upper bound is attained, is (see L'Ecuyer and Munger 2012):

$$f_\alpha^*(x_1, \ldots, x_s) = \sum_{\mathfrak{u} \subseteq \{1,\ldots,s\}} \gamma_{\mathfrak{u}} \prod_{j \in \mathfrak{u}} \frac{(2\pi)^{\alpha/2}}{(\alpha/2)!} B_{\alpha/2}(x_j). \tag{13}$$

It is also known that regardless of the choice of weights $\gamma_{\mathfrak{u}}$ (for fixed s), lattice rules can be constructed such that $\mathcal{D}_2^2(P_n)$ converges almost as fast as $n^{-\alpha}$ asymptotically when $n \to \infty$ [Sloan and Joe, 1994; Dick et al., 2006; Sinescu and L'Ecuyer, 2012]. Therefore, for all f such that $\mathcal{V}_{\mathfrak{u}}^2(f) \leq \gamma_{\mathfrak{u}}^2 < \infty$ for each $\mathfrak{u}$, the same convergence rate can be obtained for the variance.

The square variations $\mathcal{V}_{\mathfrak{u}}^2(f)$ cannot be computed explicitly in most practical applications, so to choose the weights $\gamma_{\mathfrak{u}}$, the variability of the integrand along each projection must be approximated by making certain assumptions on the structure of the integrand [Wang and Sloan, 2006; Wang, 2007; Kuo et al., 2011, 2012; L'Ecuyer and Munger, 2012]. In any case, when searching for good rules, the weights should account for the fact that more variance on $\hat{\mu}_{n,\mathrm{RQMC}}$ is built from a poor distribution of the points in certain projections of P_n, those with the larger square variations, than in others. In particular, if f is the sum of two functions that depend on disjoint sets $\mathfrak{u}$ and $\mathfrak{v}$ of variables, i.e., $f(\mathbf{x}) = f_{\mathfrak{u}}(\mathbf{x}) + f_{\mathfrak{v}}(\mathbf{x})$ with $\mathfrak{u} \cap \mathfrak{v} = \varnothing$, projections that comprise coordinates from both sets $\mathfrak{u}$ and $\mathfrak{v}$ cannot contribute any variance on $\hat{\mu}_{n,\mathrm{RQMC}}$, so their individual weights should be set to zero for $\mathcal{D}_2^2(P_n)$ to be representative of $\mathrm{Var}[\hat{\mu}_{n,\mathrm{RQMC}}]$.

3.1.4 The $\mathcal{R}_\alpha$ criterion

The $\mathcal{R}_\alpha$ criterion [Niederreiter, 1992b; Hickernell and Niederreiter, 2003] has the same structure as (8), but with $p_\alpha(x)$ replaced with

$$r_{\alpha,n}(x) = \sum_{h=-\lfloor (n-1)/2 \rfloor}^{\lfloor n/2 \rfloor} \max(1, |h|)^{-\alpha} e^{2\pi i h x} - 1. \tag{14}$$

Note the dependence of $r_{\alpha,n}$ on n. For even n, the sum in (14) is over $h = -n/2 + 1, \ldots, n/2$; for odd n, the sum is over $h = -(n-1)/2, \ldots, (n-1)/2$.

The bound (10) holds for $\mathcal{R}_\alpha$, but only if the integrand f has Fourier coefficients

$$\hat{f}(\mathbf{h}) = \int_{[0,1)^s} f(\mathbf{x})\, e^{-2\pi\sqrt{-1}\mathbf{h}\cdot\mathbf{x}}\, \mathrm{d}\mathbf{x},$$

defined for $\mathbf{h} \in \mathbb{Z}^s$, that vanish for $\mathbf{h} \notin (-n/2, n/2]^s \cap \mathbb{Z}^s$. This might be quite restrictive. On the other hand, the bound holds and can be computed for any $\alpha \geq 0$, while for the $\mathcal{P}_\alpha$ criterion it holds only for $\alpha > 1$ and the criterion can be computed directly with (8) and (9) only when α is an even integer. For other values of α, we have no formula to compute $\mathcal{P}_\alpha$ exactly and efficiently. Although the bound (10) with $\mathcal{R}_\alpha$ and $\alpha > 1$ holds for a smaller class of functions than with $\mathcal{P}_\alpha$ (for the same α), upper bounds on $\mathcal{P}_\alpha$ can be derived in terms of $\mathcal{R}_\alpha$, which is itself bounded by $\mathcal{R}_1$ [Niederreiter, 1992b; Hickernell and Niederreiter, 2003]. The rationale is that, for fixed $\alpha > 1$, extending the sum in (14) to all of $\mathbb{Z}$ only adds a "limited" contribution to the figure of merit when n is "large enough" (bounds on that contribution are given in the above references and the question of how large n should be for the contribution to be negligible is investigated empirically in Section 6.5 below); it depends very much on s, α, the generating vector, and the choice of weights. Moreover, other well-known measures of non-uniformity, such as the (weighted) star discrepancy, can also be bounded in terms of $\mathcal{R}_1$. This makes $\mathcal{R}_1$ a very general criterion, in some sense.

To evaluate $\mathcal{R}_\alpha$, only the values of $r_{\alpha,n}(x)$ evaluated at $x = i/n$ for $i = 0, \ldots, n-1$ are needed. These can be efficiently calculated, through the use of fast Fourier transforms (FFT), as follows (based on an idea suggested to us by Fred Hickernell). First, we need to rewrite (14) in the form of a discrete Fourier transform. To do so, we replace h with $h - n$ in the part of the sum that is over the negative values of h:

$$r_{\alpha,n}(x) = \sum_{h=0}^{n-1} \hat{r}_{\alpha,n}(h)\, e^{2\pi i h x},$$

where

$$\hat{r}_{\alpha,n}(h) = \begin{cases} 0 & \text{if } h = 0 \\ h^{-\alpha} & \text{if } 0 < h \leq n/2 \\ (n-h)^{-\alpha} & \text{if } n/2 < h < n. \end{cases}$$

The values of $r_{\alpha,n}(i/n)$ for $i = 0, \ldots, n-1$ are given by the n-point discrete Fourier transform of $\hat{r}_{\alpha,n}(h)$ at $h = 0, \ldots, n-1$, and can be calculated with an FFT. This is how it is done in Lattice Builder.

3.1.5 Criteria based on the spectral test

Another type of projection-dependent figure of merit supported by Lattice Builder is based on the spectral test, as in L'Ecuyer and Lemieux [2000]. The projection of the lattice L_s onto the coordinates in $\mathfrak{u}$ has its points arranged in a family of equidistant parallel hyperplanes in $\mathbb{R}^{|\mathfrak{u}|}$. Let $\mathcal{L}_\mathfrak{u}(P_n) = \mathcal{L}_\mathfrak{u}(P_n(\mathfrak{u}))$ denote the distance between the successive parallel hyperplanes, maximized over all parallel hyperplane families that cover all the points. This distance is computed as in L'Ecuyer and Couture [1997]. We define the *spectral* figure of merit as

$$\mathcal{D}_\mathfrak{u}(P_n) = \frac{\mathcal{L}_\mathfrak{u}(P_n)}{\mathcal{L}^*_{|\mathfrak{u}|}(n)} \geq 1, \tag{15}$$

where $\mathcal{L}^*_{|\mathfrak{u}|}(n)$ is a lower bound on the distance $\mathcal{L}_\mathfrak{u}(P_n)$ that depends only on $|\mathfrak{u}|$ and n, obtained from lattice theory [Conway and Sloane, 1999; L'Ecuyer, 1999a; L'Ecuyer and Lemieux, 2000]. This figure of merit (15) cannot be smaller than 1 and we want to minimize it. The normalization in (15) permits one to consistently compare the merit values of the projections $P_n(\mathfrak{u})$ of different orders $|\mathfrak{u}|$. In Lattice Builder, user-defined normalization constants can also replace the bounds $\mathcal{L}^*_{|\mathfrak{u}|}(n)$. L'Ecuyer and Lemieux [2000] used $\mathcal{D}_\mathfrak{u}(P_n)$ as given by (15) together with $q = \infty$ in (4) with a unit weight assigned to a selection of projections, and zero weight to others. Equation (4) used with (15) constitutes, in a sense, an approximation to the right-hand side of (11) [L'Ecuyer and Lemieux, 2000], with different conditions on f. The weights should still reflect the relative magnitude of the contributions associated to the different projections of the point set to the variance of the RQMC estimator.

3.2 Types of weights

Under some special configurations, the $2^s - 1$ projection-dependent weights $\gamma_{\mathfrak{u}}$ can be expressed in terms of fewer than $2^s - 1$ independent parameters, and this allows for a more efficient evaluation of certain types of figures of merit, as will be explained in Section 3.3 below. In Lattice Builder, the weights can be specified for individual projections and default weights can be applied to groups of projections of the same dimension. Lattice Builder also implements three special classes of weights known as order-dependent weights, product weights, as well as product and order-dependent (POD) weights.

The weights are called *order-dependent* when all projections $P_n(\mathfrak{u})$ of the same order $|\mathfrak{u}|$ have the same weight, i.e., when there exists non-negative constants $\Gamma_1, \dots, \Gamma_s$ such that $\gamma_{\mathfrak{u}} = \Gamma_{|\mathfrak{u}|}$ for $\varnothing \neq \mathfrak{u} \subseteq \{1, \dots, s\}$ [Dick et al., 2006]. If $\Gamma_k \neq 0$ and $\Gamma_j = 0$ for all $j > k$, the order-dependent weights are said to be *finite-order* of order k. In Lattice Builder, a weight can be specified explicitly for the first few projection orders, then a default weight can be specified for higher orders.

For *product weights* [Hickernell, 1998a,b; Sloan and Woźniakowski, 1998], each coordinate $j = 1, \dots, s$ is assigned a non-negative weight γ_j such that $\gamma_{\mathfrak{u}} = \prod_{j \in \mathfrak{u}} \gamma_j$, for $\varnothing \neq \mathfrak{u} \subseteq \{1, \dots, s\}$. As for the order-dependent weights, Lattice Builder allows the user to specify explicit weights for the first few coordinates, then to set a default weight for the rest of them.

POD weights [Kuo et al., 2011, 2012], a hybrid between product weights and order-dependent weights, are of the form $\gamma_{\mathfrak{u}} = \Gamma_{|\mathfrak{u}|} \prod_{j \in \mathfrak{u}} \gamma_j$. They can be specified in Lattice Builder as would be product weights and order-dependent weights together.

Lattice Builder allows the user to specify the q-th power of the weights, $\gamma_{\mathfrak{u}}^q$, as a sum of the q-th power of weights of different types. Thus, it is possible, for instance, to choose order-dependent weights as a basis, and to add more weight to a few selected projections by specifying projection-dependent weights for these, on top of the order-dependent weights. Besides, arbitrary special cases of the projection-dependent weights can be implemented in Lattice Builder through the API.

When adding weights of different structures together, the corresponding sum in (4) can be separated into multiple sums, one corresponding to each type of weights. The software thus computes the figure of merit for each type of weights separately, using evaluation algorithms specialized for each of them, then sums the individual results to produce the total figure of merit. One could think of multiplying (instead of adding) weights of different structures, like POD weights result from multiplying product and order-dependent weights, but then the resulting sum in (4) cannot be easily decomposed into simpler sums that we know how to evaluate. This requires implementing new evaluation algorithms, as done for POD weights.

3.3 Weighted coordinate-uniform figures of merit

In this section, we consider $\mathcal{D}_q$ defined as in (4) and with $\mathcal{D}_{\mathfrak{u}}^q$ as in (8), where p_α is replaced by any function $\omega : [0, 1) \to \mathbb{R}$:

$$[\mathcal{D}_{\mathfrak{u}}(P_n)]^q = \frac{1}{n} \sum_{i=0}^{n-1} \prod_{j \in \mathfrak{u}} \omega((i a_j / n) \bmod 1). \tag{16}$$

We call this $\mathcal{D}_q$ a *coordinate-uniform* figure of merit. The software implements (16) for $\omega = p_\alpha$ as in (9) and for $\omega = r_{\alpha,n}$ as in (14). These choices respectively yield the $\mathcal{P}_\alpha$ and $\mathcal{R}_\alpha$ criteria when $q = 2$, but do not correspond to known criteria for other values of q. To avoid any confusion, the software allows only $q = 2$ when using the evaluations methods described in the following paragraphs for (16). Note that choosing any other value of q just raises the final value of the figure of merit to the power $2/q$, and does not change the ranking of lattice rules.

We introduce algorithms that compute figures of merit as in (16) more efficiently than to independently compute the $2^s - 1$ terms in (4). For each type of weights from Section 3.2, there is a specialized algorithm, described below.

3.3.1 Storing vs. recomputing

Let $\omega = (\omega_0, \ldots, \omega_{n-1})$ be the vector with components $\omega_i = \omega(i/n)$ for $i = 0, \ldots, n-1$. For $j = 1, \ldots, s$, also let $\omega^{(j)} = (\omega_{\pi_j(0)}, \ldots, \omega_{\pi_j(n-1)})$ denote vector ω resulting from permutation $\pi_j(i) = ia_j \bmod n$ for $i = 0, \ldots, n-1$. (Note that π_j is a permutation only if a_j and n are coprime, which we have already assumed in Section 2.2.) The last term on the right-hand side of (7) can thus be written in vector form as

$$\frac{1}{n}\sum_{i=0}^{n-1}\omega_{\pi_j(i)}\sum_{\mathfrak{u}\subseteq\{1,\ldots,j-1\}}\gamma^q_{\mathfrak{u}\cup\{j\}}\prod_{k\in\mathfrak{u}}\omega_{\pi_k(i)} = \frac{\omega^{(j)}}{n}\bullet\left(\sum_{\varnothing\neq\mathfrak{u}\subseteq\{1,\ldots,j-1\}}\gamma^q_{\mathfrak{u}\cup\{j\}}\mathbf{q}_{\mathfrak{u}}\right)$$

where $\bullet$ denotes the scalar product, and

$$\mathbf{q}_{\mathfrak{u}} = \left(\prod_{k\in\mathfrak{u}}\omega_{\pi_k(0)},\ \ldots,\ \prod_{k\in\mathfrak{u}}\omega_{\pi_k(n-1)}\right) = \bigodot_{k\in\mathfrak{u}}\omega^{(k)},$$

where $\odot$ denotes the element-by-element product, e.g. $(x_1, \ldots, x_s) \odot (y_1, \ldots, y_s) = (x_1y_1, \ldots, x_sy_s)$. Putting everything together, we obtain, for $j = 1, \ldots, s$,

$$\mathcal{D}^q_{q,j}(P_n) = \mathcal{D}^q_{q,j-1}(P_n) + \frac{\omega^{(j)}\bullet\bar{\mathbf{q}}_j}{n} \tag{17}$$

$$\bar{\mathbf{q}}_j = \sum_{\mathfrak{u}\subseteq\{1,\ldots,j-1\}}\gamma^q_{\mathfrak{u}\cup\{j\}}\,\mathbf{q}_{\mathfrak{u}} \tag{18}$$

$$\mathbf{q}_{\mathfrak{u}\cup\{j\}} = \omega^{(j)}\odot\mathbf{q}_{\mathfrak{u}} \qquad (\mathfrak{u}\subseteq\{1,\ldots,j-1\}), \tag{19}$$

with initial states $\mathcal{D}_{q,0}(P_n) = 0$ and $\mathbf{q}_{\varnothing} = \mathbf{1}$. Assuming that $\mathbf{q}_{\mathfrak{u}}$ is already available for $\mathfrak{u}\subseteq\{1,\ldots,j-1\}$, computing $\bar{\mathbf{q}}_j$ requires $\mathcal{O}(2^jn)$ operations and storage for all states $\mathbf{q}_{\mathfrak{u}}$ for $\mathfrak{u}\subseteq\{1,\ldots,j-1\}$.

For comparison, computing separately the 2^j terms in the sum on right-hand side of (7) in coordinate-uniform form requires constant storage and $\mathcal{O}(2^jjn)$ operations, as explained below. The terms in the sum on the right-hand side of (7) can be regrouped by projection order $\ell = |\mathfrak{u}|$, ranging from 0 to $j-1$. There are $\binom{j-1}{\ell}$ projections of order ℓ, and evaluating $\mathcal{D}^q_{\mathfrak{u}\cup\{j\}}$ of the form of (16) requires the multiplication of $\ell+1$ factors and the addition of n terms. Hence, evaluating all terms on the right-hand side of (7) requires a number of operations of the order of

$$\sum_{\ell=0}^{j-1}\binom{j-1}{\ell}(\ell+1)n = 2^{j-2}(j+1)n$$

because $\sum_{\ell=0}^{j}\binom{j}{\ell}\ell = 2^{j-1}j$. This complexity analysis applies to general projection-dependent weights. Simplifications can be made for special types of weights, as discussed in Section 3.3.3 below.

Overall, this second approach takes only $\mathcal{O}(n)$ space for the precomputed values $\omega(i/n)$, so it needs less storage than the first, but more operations. The user may select one of these two approaches depending on the situation. In the case of ordinary lattices, Lattice Builder stores only ω; the components of $\omega^{(j)}$ are obtained dynamically by applying the permutation π_j defined above to the components of ω.

3.3.2 Embedded lattices

For embedded lattices in base b, a distinct merit value must be computed for each level. So, we define a distinct vector ω_k of length $(b-1)b^{k-1}$ for each level $k = 1, \ldots, m$, whose i-th component has value $\omega(\eta_b(i)/b^k)$, where the mapping

$$\eta_b(i) = i + \lfloor (i-1)/(b-1)\rfloor$$

simply skips all multiples of b. For example, with $b = 3$ and $k = 2$, we have

$$\omega_2 = (\omega(1/9), \omega(2/9), \omega(4/9), \omega(5/9), \omega(7/9), \omega(8/9))\,.$$

For $k = 0$, we define $\omega_0 = (\omega(0))$. So, for embedded lattices, Lattice Builder stores

$$\omega = (\omega_0, \dots, \omega_m)$$

as the concatenation of the subvectors ω_k for all individual levels $k = 0, \dots, m$. The other state vectors $\omega^{(j)}$, $\bar{\mathbf{q}}_j$ and $\mathbf{q}_{\mathfrak{u}\cup\{j\}}$ are defined accordingly, which allows for standard vector operations (addition, multiplication by a scalar) to be carried out on all levels at the same time, transparently, using the same syntax as for ordinary lattices. Note that the first b^k components of a vector contain all the information relative to level k, so the scalar product at level k between two vectors $\mathbf{x} = (x_1, \dots, x_{b^m})$ and $\mathbf{y} = (y_1, \dots, y_{b^m})$, which we denote by $[\mathbf{x} \bullet \mathbf{y}]_k$, can be obtained by computing the scalar product using exclusively the first b^k components of each vector:

$$[\mathbf{x} \bullet \mathbf{y}]_k = \sum_{i=1}^{b^k} x_i y_i = [\mathbf{x} \bullet \mathbf{y}]_{k-1} + \sum_{i=b^{k-1}+1}^{b^k} x_i y_i.$$

where the second equality holds for $k \geq 1$. It follows that the scalar products for all levels can be computed incrementally by reusing the results for the lower levels. For example, the scalar product at level $k \geq 1$ can be obtained by adding the scalar product using only the components of indices $b^{k-1} + 1$ to b^k to the result of the scalar product at level $k - 1$.

3.3.3 Special types of weights

Eqs. (18) and (19) generalize to projection-dependent weights the recurrence formulas previously obtained by Cools et al. [2006] and Nuyens and Cools [2006a] for the simpler cases of order-dependent and product weights, respectively. In these cases, (17) still holds but the definitions of $\bar{\mathbf{q}}_j$ and of the state vectors are different from (18) and (19). For order-dependent weights and $j = 1, \dots, s$, we have

$$\bar{\mathbf{q}}_j = \sum_{\ell=0}^{j-1} \Gamma^q_{\ell+1} \, \mathbf{q}_{j-1,\ell} \tag{20}$$

$$\mathbf{q}_{j,\ell} = \mathbf{q}_{j-1,\ell} + \omega^{(j)} \odot \mathbf{q}_{j-1,\ell-1} \qquad (\ell = 1, \dots, j), \tag{21}$$

with $\mathbf{q}_{j,0} = \mathbf{1}$ for all j and $\mathbf{q}_{j,\ell} = \mathbf{0}$ where $\ell > j$. In practice, we overwrite $\mathbf{q}_{j-1,\ell}$ with $\mathbf{q}_{j,\ell}$ when j is increased. However, $\mathbf{q}_{j-1,\ell-1}$ must still be available when computing $\mathbf{q}_{j,\ell}$, so for fixed j, we proceed by decreasing order of ℓ. Assuming that $\mathbf{q}_{j-1,\ell}$ is already available for $\ell = 1, \dots, j-1$, computing $\bar{\mathbf{q}}_j$ with order-dependent weights requires $\mathcal{O}(jn)$ operations and storage only for the states $\mathbf{q}_{j-1,\ell}$ for $\ell = 1, \dots, j-1$.

For product weights and $j = 1, \dots, s$, we have

$$\bar{\mathbf{q}}_j = \gamma^q_j \mathbf{q}_{j-1} \tag{22}$$

$$\mathbf{q}_j = \left(\mathbf{1} + \gamma^q_j \omega^{(j)}\right) \odot \mathbf{q}_{j-1}, \tag{23}$$

with $\mathbf{q}_0 = \mathbf{1}$. Assuming that $\mathbf{q}_{j-1}$ is already available, computing $\bar{\mathbf{q}}_j$ with product weights requires $\mathcal{O}(jn)$ operations and $\mathcal{O}(n)$ storage only for the state $\mathbf{q}_{j-1}$.

The approach for POD weights was proposed by Kuo et al. [2012] and turns out to be a slight modification of the case of order-dependent weights. We have

$$\bar{\mathbf{q}}_j = \gamma^q_j \sum_{\ell=0}^{j-1} \Gamma^q_{\ell+1} \, \mathbf{q}_{j-1,\ell} \tag{24}$$

$$\mathbf{q}_{j,\ell} = \mathbf{q}_{j-1,\ell} + \gamma^q_j \, \omega^{(j)} \odot \mathbf{q}_{j-1,\ell-1} \qquad (\ell = 1, \dots, j), \tag{25}$$

with $\mathbf{q}_{j,0} = \mathbf{1}$ for all j and $\mathbf{q}_{j,\ell} = \mathbf{0}$ where $\ell > j$. The storage and algorithmic complexities are unchanged.

When the user specifies a sum of weights of different types, Lattice Builder, first evaluates the coordinate-uniform figure of merit for each type of weights using the appropriate specialized algorithm, then adds up the computed merit values.

Regardless of the type of weights, if $a_1, \dots, a_{j-1}$ are kept untouched, the same weighted state $\bar{\mathbf{q}}_j$ and prior merit value $\mathcal{D}_{q,j-1}(P_n)$ can be used to compute $\mathcal{D}_{q,j}(P_n)$ for different values of a_j, which makes the CBC algorithms described in Section 4 very efficient.

Specialized evaluation algorithms for custom types of weights can be implemented in Lattice Builder simply by redefining (18) and (19) through the API.

3.4 Transformations and filters

In Lattice Builder, it is possible to configure a chain of transformations and filters that will be applied to the computed values of a figure of merit. To every original merit value $\mathcal{D}_q(P_n)$ in (4), the transformation associates a transformed merit value $\mathcal{E}_q(P_n)$. For example, the transformed value may have the form

$$\mathcal{E}_q(P_n) = \frac{\mathcal{D}_q(P_n)}{D_q^*(n)}$$

where $D_q^*(n)$ is some normalization factor, e.g., a bound on (or an estimate of) the best possible value of $\mathcal{D}_q(P_n)$, or a bound on the average of $\mathcal{D}_q(P_n)$ over all values of $a_1, \dots, a_s$ under consideration, for the given n and s. Such transformations can be useful for example to define comparable measures of uniformity for lattices of different sizes n and combining them to measure the global quality of a set of embedded lattices (see Section 3.5) or, when constructing lattices by some random search procedure, to eliminate lattices whose normalized figure of merit (e.g., for the projections over the j coordinates selected so far, in the case of a CBC construction) is deemed too poor. In the latter case, a filter can be applied after the transformation to exclude the candidate vectors $\mathbf{a}$ for which $\mathcal{E}_q(P_n)$ exceeds a given threshold. Such acceptance-rejection schemes were proposed and used by L'Ecuyer and Couture [1997], Wang and Sloan [2006], and Dick et al. [2008], among others.

For $\mathcal{P}_\alpha$ with projection-dependent weights and $q = 2$, Lattice Builder implements the bound derived by Sinescu and L'Ecuyer [2012], valid for any $\lambda \in [1/\alpha, 1)$:

$$[D_2(P_n)]^2 \le [D_2^*(n;\lambda)]^2 = \left[\frac{1}{\varphi(n)} \sum_{\varnothing \ne \mathfrak{u} \subseteq \{1,\dots,s\}} \gamma_{\mathfrak{u}}^{2\lambda} \, (2\zeta(\lambda\alpha))^{|\mathfrak{u}|}\right]^{1/\lambda}, \tag{26}$$

where φ is Euler's totient function and ζ is the Riemann zeta function. For ordinary lattice rules, Lattice Builder selects $D_2^*(n) = D_2^*(n;\lambda^*)$, where λ^* minimizes $D_2^*(n;\lambda)$ over the interval $[1/\alpha, 1)$. We also have specialized its expression for more efficient computation with order-dependent weights:

$$[D_2^*(n;\lambda)]^2 = \left[\frac{1}{\varphi(n)} \sum_{\ell=1}^{s} \Gamma_\ell^{2\lambda} \, y_{s,\ell}(\lambda)\right]^{1/\lambda}, \tag{27}$$

where

$$y_{s,\ell}(\lambda) = \frac{s-\ell+1}{\ell} \, 2\zeta(\alpha\lambda) \, y_{s,\ell-1}(\lambda)$$

for $\ell = 1, \dots, s$, with $y_{s,0}(\lambda) = 1$. For product weights, we have:

$$[D_2^*(n;\lambda)]^2 = \frac{1}{\varphi(n)^{1/\lambda}} \left[\prod_{j=1}^{s} \left(1 + 2\gamma_j^{2\lambda} \, \zeta(\alpha\lambda)\right) - 1\right]^{1/\lambda}. \tag{28}$$

For POD weights, it is easily seen that (27) still holds, but with:

$$y_{j,\ell}(\lambda) = y_{j-1,\ell}(\lambda) + 2\gamma_j^{2\lambda} \zeta(\alpha\lambda) \, y_{j-1,\ell-1}(\lambda)$$

for $\ell = 1, \dots, j$, with $y_{j,0}(\lambda) = 1$ for all j and $y_{j,\ell}(\lambda) = 0$ when $\ell > j$. As was the case with (21), the $y_{j-1,\ell}$ here can be overwritten with $y_{j,\ell}$, as long as we proceed by decreasing order of ℓ. Lattice Builder deals with

sums of weights of different types by decomposing the sum in (26) into multiple sums, one corresponding to each type of weights. This allows for using the efficient implementations specialized for each type of weights.

As an alternative to (28), Lattice Builder also implements the bound derived by Dick et al. [2008] for product weights, also valid only for $q = 2$ and $\lambda \in [1/\alpha, 1)$:

$$[D_2^*(n;\lambda)]^2 = \frac{1}{n^{1/\lambda}} \left[\prod_{j=1}^{s} \left(1 + 2^{\kappa+1} \gamma_j^{2\lambda}\, \zeta(\lambda\alpha)\right) - 1 \right]^{1/\lambda}, \tag{29}$$

where κ is the number of distinct prime factors of n.

Arbitrary transformations and filters can also be defined by the user through the API. By default, Lattice Builder applies no transformation, i.e., $\mathcal{E}_q(P_n) = \mathcal{D}_q(P_n)$.

3.5 Weighted multi-level figures of merit

For embedded lattices, a distinct normalized merit value $\mathcal{E}_q(P_{n_k})$ must be associated to each embedded level $k = 1, \ldots, m$. To produce a global scalar merit value $\bar{\mathcal{E}}_q(P_{n_1}, \ldots, P_{n_m})$ for the set of embedded lattices, a first available approach is to normalize the merit values at each level k as explained in Section 3.4, then combine these normalized values by taking the worst (largest) weighted value [Hickernell et al., 2001; Cools et al., 2006]:

$$\left[\bar{\mathcal{E}}_q(P_{n_1}, \ldots, P_{n_m})\right]^q = \max_{k=1,\ldots,m} w_k \left[\mathcal{E}_q(P_{n_k})\right]^q \tag{30}$$

where w_k is the non-negative per-level weight for $k = 1, \ldots, m$. A second available approach is to compute the weighted average [Dick et al., 2008] instead of taking the maximum:

$$\left[\bar{\mathcal{E}}_q(P_{n_1}, \ldots, P_{n_m})\right]^q = \sum_{k=1}^{m} w_k \left[\mathcal{E}_q(P_{n_k})\right]^q. \tag{31}$$

As a special case, one may put $w_m = 1$ and $w_k = 0$ for $k < m$, which amounts to using the algorithms for embedded lattices to construct an ordinary lattice of size n_m. The fast CBC construction algorithm [Cools et al., 2006], discussed in Section 4.4 below, is actually implemented in this way in Lattice Builder. The implementation requires that the number of points has the form $n = b^m$ with b prime (although fast CBC could also be implemented for arbitrary composite n), and the algorithm must compute merit values for embedded levels anyway. Custom methods of combining multi-level merit values can also be defined by the user through the API.

For embedded lattices constructed using $\mathcal{P}_\alpha$ for the projections and with $q = 2$, Lattice Builder selects $D_2^*(n_k) = D_2^*(n_k; \lambda^*)$ at level k, where λ^* minimizes $c_k^{-1/\lambda} [D_2^*(n_k;\lambda)]^2$, with $D_2^*(n_k;\lambda)$ selected among the bounds (26) to (29), and where each constant $c_k \geq 0$, which must be specified by the user, defines a weight $w_k = c_k^{1/\lambda^*}$ at level k. Dick et al. [2008] proved that with the CBC construction algorithm (see Section 4), if we normalize with the bound (29) and filter out all candidate lattices for which $\mathcal{E}_2(P_{n_k}) > 1$ at any level k, the algorithm is guaranteed to return an embedded lattice rule for which $\mathrm{Var}[\hat{\mu}_{n_k,\mathrm{RQMC}}]$ converges at a rate arbitrarily close to $\mathcal{O}(n_k^{-\alpha})$ for all levels k for which $c_k > 0$.

4 Construction methods

Recall that our search for lattice rules is restricted to the space of fully projection-regular rank-1 integration lattices. This means that for any given n, we want to consider only the generating vectors $\mathbf{a} = (a_1, \ldots, a_s)$ whose coordinates are all relatively prime with n. We may want to enumerate all those vectors to perform an exhaustive search, or enumerate only those having a given form (e.g., to search among Korobov lattices), or just sample a few of them at random. In this section, we discuss the techniques that we have designed and implemented to perform this type of enumeration or sampling. Then, we review different construction methods supported by Lattice Builder. For all these methods, filters are applied as part of the construction process for early rejection of candidate lattices as soon as their normalized merit value exceeds a given threshold.

4.1 Enumerating the integers coprime with n

For any $n > 1$, let

$$U_n = \{i \in \mathbb{Z}_n^* \,:\, \gcd(i, n) = 1\},$$

which can be identified with $(\mathbb{Z}/n\mathbb{Z})^\times$, the multiplicative group of integers modulo n. Its cardinality is given by Euler's totient function $\varphi(n)$. For all construction methods, we only consider values $a_j \in U_n$. Without loss of generality, we also assume that $a_1 = 1$, for simplicity.

A naive enumeration of the $\varphi(n)$ elements in U_n could be done by going through each of the $n-1$ elements of $\mathbb{Z}_n^*$ and filtering out those that are not coprime with n by computing their greatest common divisors. In practice, the elements of U_n often need to be enumerated repeatedly, depending on the search space for $\mathbf{a}$ and the construction method. An alternative approach would be to enumerate them once for all and store them in an array. But this is not efficient if n is large and we only want to pick a few of them at random. The algorithm in Lattice Builder that enumerates the elements of U_n assigns them indices based on their rank, using the Chinese remainder theorem, as explained below. It permits one to pick random elements efficiently by randomly selecting indices from 1 to $\varphi(n)$.

Suppose n is a composite integer that can be factored as $n = \nu_1\nu_2\dots\nu_\ell$, where $\nu_j = \beta_j^{\mu_j}$ for $j = 1, \dots, \ell$, and where $\beta_1 < \cdots < \beta_\ell$ are ℓ distinct prime factors with respective integer powers $\mu_1, \dots, \mu_\ell$. The Chinese remainder theorem states that there is an isomorphism between $\mathbb{Z}_n^*$ and $Z_n^* = \mathbb{Z}_{\nu_1}^* \times \cdots \times \mathbb{Z}_{\nu_\ell}^*$ that maps $k \in \mathbb{Z}_n^*$ to $\kappa = (\kappa_1, \dots, \kappa_\ell) \in Z_n^*$. Here, we set $\kappa_j = k \bmod \nu_j$ for $j = 1, \dots, \ell$. Notice that k is coprime with n if and only if $\kappa_j \bmod \beta_j \neq 0$ for all j. The converse mapping can be verified to be

$$k = \left[\sum_{j=1}^{\ell} (n/\nu_j)\kappa_j\xi_j\right] \bmod n, \tag{32}$$

where ξ_j is the multiplicative inverse of n/ν_j modulo ν_j [Knuth, 1998, Section 4.3.2]. These ξ_j can be obtained with the extended Euclidean algorithm. In Lattice Builder, given n, all the constants n/ν_j and ξ_j for $j = 1, \dots, \ell$ are computed in advance and stored for subsequent use. To enumerate the elements of U_n, all values of $\kappa \in Z_n^*$ composed exclusively of nonzero components are enumerated (this is a straightforward process), then mapped through (32). The resulting ordering of U_n is increasing if n is prime, but not necessarily otherwise.

When n is a power of a prime, the multiplicative group $(\mathbb{Z}/n\mathbb{Z})^\times$ is cyclic. The fast CBC algorithm described in Section 4.4 requires a special ordering of U_n in which the i-th element is $g^i \bmod n$, where g is the smallest generator of this group; see Cools et al. [2006]. If $n \geq 4$ is a power of 2, we use $g = 5$ to generate the first half of the group, then the second half can be obtained by multiplying the first by -1, modulo n.

4.2 Arbitrary traversal method

The enumeration methods described in Section 4.1 define an implicit ordering of the elements of U_n. In some cases, we just want to enumerate the elements of U_n in this order. This is called *forward traversal* in Lattice Builder. In other situations, we may wish to draw a certain number of values at random from U_n. Then, Lattice Builder randomly picks indices for the elements of U_n, from the set $\{1, \dots, \varphi(n)\}$. This is called *random traversal*. This means that Lattice Builder never misses when randomly picking an integer coprime with n. Other arbitrary traversal methods could be defined through the API.

To illustrate the idea, suppose we want to search for lattice rules with $n = 1024$, so the coordinates of $\mathbf{a}$ must belong to U_{1024}. The following chunk of C++ code instantiates a list (or sequence) called `myGenSeq` of all integers in U_{1024}, in the order given by the above enumeration method:

```
typedef Traversal::Forward MyTraversal;
typedef GenSeq::CoprimeIntegers<Compress::NONE, MyTraversal> MyGenSeq;
MyGenSeq myGenSeq(1024, MyTraversal());
```

The first line defines the type alias `MyTraversal` for the forward traversal method. The second line defines the type alias `MyGenSeq` as a type of sequence of all integers in U_n, to be enumerated by the forward traversal

method. The `Compress::NONE` token means that the compression discussed in the next subsection is not applied. The last line instantiates such a sequence with $n = 1024$ and an instance of the forward traversal method. The following C++11 code outputs all the values contained in `myGenSeq`, i.e., the list of all odd numbers from 1 to 1023.

```
for (auto value : myGenSeq)
  std::cout << value << std::endl;
```

Recall that Lattice Builder does not actually store these values; they are generated on-the-fly using the methods from Section 4.1. To randomly draw 10 values from U_n instead of listing them all, we can use:

```
typedef Traversal::Random<LFSR113> MyTraversal;
typedef GenSeq::CoprimeIntegers<Compress::NONE, MyTraversal> MyGenSeq;
MyGenSeq myGenSeq(1024, MyTraversal(10));
```

The only changes from the previous piece of code are that the type alias `MyTraversal` now corresponds to the random traversal method, using the pseudo-random generator LFSR113 of L'Ecuyer [1999b], and the number of values to draw (10) is given in the instantiation on the last line.

4.3 Symmetric compression

If the projection-dependent figure of merit under consideration is symmetric, i.e., is invariant under the transformation $a_j \mapsto n - a_j$ for any $j = 1, \dots, s$, then the search space for each component can be reduced by half. This happens for instance in the case of coordinate-uniform figures of merit such that $\omega(x) = \omega(1 - x)$ for $x \in [0, 1)$. For an exhaustive search, this can reduce the search space by a factor of 2^{s-1}. This idea was used by Cools et al. [2006] in their fast CBC algorithm. It is called *symmetric compression* in Lattice Builder: it reduces to half the number of output values from U_n, and maps each value $k \in U_n$ to $\min(k, n - k)$.

4.4 Construction methods currently supported

The following construction methods are currently implemented. For all these methods, we try to find a generating vector $\mathbf{a}$ with the smallest possible value of $\mathcal{E}_q(P_n)$ for ordinary lattices or of $\bar{\mathcal{E}}_q(P_{n_1}, \dots, P_{n_m})$ for embedded lattices.

Exhaustive search An exhaustive search examines all generating vectors $\mathbf{a} \in U_n^s$ and retains the best.

Random search over all possibilities In this variant, we draw r random values of $\mathbf{a}$ uniformly distributed in U_n^s and retain the best.

Korobov construction This is a variant of the exhaustive search that considers only the vectors $\mathbf{a}$ of the form $\mathbf{a} = (1, a, a^2 \bmod n, \dots, a^s \bmod n)$ for $a \in U_n$.

Random Korobov construction This is a variant of the Korobov construction that considers only r random values of a uniformly distributed in U_n.

CBC construction The component-by-component (CBC) algorithm [Kuo and Joe, 2002; Dick et al., 2006] constructs the vector $\mathbf{a}$ one coordinate at a time, as follows. It starts with $a_1 = 1$. Then for $j = 2, \dots, s$, assuming that $a_1, \dots, a_{j-1}$ have been selected and are now kept fixed, the algorithm selects the value of $a_j \in U_n$ that minimizes $\mathcal{E}_q(P_n(\{1, \dots, j\}))$ for ordinary lattices or $\bar{\mathcal{E}}_q(P_{n_1}(\{1, \dots, j\}), \dots, P_{n_m}(\{1, \dots, j\}))$ for embedded lattices. That is, at step j, we minimize the figure of merit for the points truncated to their first j coordinates.

Random CBC construction This is a variant of the CBC algorithm where, for $j = 2, \dots, s$, instead of considering for a_j all $a \in U_n$, only r random values of a uniformly distributed in U_n are considered [Wang and Sloan, 2006; Sinescu and L'Ecuyer, 2009].

Fast CBC construction This method applies for the figures of merit of the form described in Section 3.3. It is implemented only for n equal to a power of a prime number (including 1). It uses a variant of the CBC algorithm that computes the merit values for all values of a_j simultaneously, for each fixed j, through the use of fast Fourier transforms [Cools et al., 2006; Nuyens and Cools, 2006a,c,b]. If $K_{n,j}$ denotes the cost for evaluating the figure of merit for a single value of a_j, as given in Section 3.3, then the ordinary CBC algorithm requires $\mathcal{O}(K_{n,j}n)$ operations to evaluate the merit values for all $\phi(n)$ values of a_j, while the fast CBC variant requires only $\mathcal{O}(K_{n,j}\log n)$ operations. The details are explained in the references above. Typical values of $K_{n,j}$ are $\mathcal{O}(nj)$ for product or order-dependent weights and $\mathcal{O}(n2^j)$ for general projection-dependent weights. We use the FFT implementation from the FFTW library [Frigo and Johnson, 2005].

Extending the number of points Consider embedded lattices rules in base b with maximum level m and generating vector $\mathbf{a} = (a_1, \dots, a_s)$. The maximum number of points $n_m = b^m$ can be augmented to $n_{m+1} = b^{m+1}$ by keeping the rightmost m digits in base b of each $a_1, \dots, a_s$ unchanged, while adding an $(m+1)$th digit in base b to each $a_1, \dots, a_s$, selected to minimize the figure of merit of the resulting extended n_{m+1}-point lattice. This means that there are b^s candidate lattices to consider when we increase m by 1. Note that there is (currently) no theoretical proof that the discrepancy will keep improving by doing this. The intuition is that it may work a limited number of times. A more robust approach is to construct a set of embedded lattices while making sure that m is selected large enough in the first place.

4.5 Construction layers

Constructing lattice rules in Lattice Builder consists of traversing a sequence of candidate lattices and of picking the best one(s). In fact, the search process is organized as multiple layers of sequences of different types, all connected together, starting at the lowest level with sequences of integers coprime with n (as in Section 4.1) followed by sequences of candidate lattice rules, then sequences of merit values and of filtered merit values (as explained in Sections 3.4 and 3.5) at the upper level.

For example, to perform a Korobov search in dimension 3, we define at the bottom a sequence $(1, \dots, n-1)$ of integers coprime with n, each element of which is mapped to an element from a sequence of candidate lattice rules with respective generating vectors $(1,1,1), \dots, (1, n-1, (n-1)^2 \bmod n)$. Then, on top of it is created a sequence of merit values, which maps, in the same order, each candidate lattice rule to its merit value. Finally, transformations (normalizations) and filters are applied elementwise, and the resulting values are made accessible through a sequence of transformed merit values.

In practice, the elements of a sequence are accessed through iterators. When an iterator on the top sequence (transformed merit values) is requested to return the value it points to, the request travels down to the bottom layer: the current element of the generator sequence is used to generate the current element of the sequence of generating vectors, which in turn is used to compute the current element of the sequence of merit values, before going through the chain of transformations and filters. Thus, merit values are computed on-the-fly, just as the iterator on the sequence at the top is advanced. There are exceptions where the merit values are computed in advance and stored in an array; in that case the iterator returns the values in the array instead of dynamically computed values. One such exception occurs when the fast CBC construction is used. This multilayer design allows for transparent implementations of sequences at any level as dynamically computed values, or as cached or precomputed values. In some cases, the sequence-based evaluation process of figures of merit allows for efficient reuse of values that were computed for previous elements. Continuous feedback from the construction algorithm is provided through C++ signals provided by the Boost library [Boost.org, 2012].

4.6 CBC-based non-CBC constructions

In order to avoid repetitions, the code for the non-CBC constructions, such as exhaustive or random, makes use of the CBC algorithm. For each generating vector $\mathbf{a}$ visited by these constructions, the code for the CBC

algorithm (for either the generic implementation or the specialized coordinate-uniform implementations) is applied to a singleton search space containing only $\mathbf{a}$, thus enabling the efficient specializations of the projection-dependent figures of merit in the coordinate-uniform case, at the cost, in some cases, of a larger memory usage to store the CBC state vectors from (19), (21) or (23).

4.7 Decoupling and extensibility

The construction process is defined by several components: type of lattice (Section 2.2), enumeration method for U_n (Section 4.1), traversal type (Section 4.2), type of projection-dependent figure of merit and of accumulator (Section 3.1), type of weights (Section 3.2), construction method (Section 4.4), transformations and filters (Section 3.4), and combining methods for embedded lattices (Section 3.5). Lattice Builder was designed to keep these components decoupled, using modern software design methods [Alexandrescu, 2001; Lischner, 2009]. This facilitates code reuse while providing the necessary flexibility in combining pieces. In addition to the set of components already implemented, the API encourages the user to extend Lattice Builder by implementing custom types of figures of merit, weights, filters, etc.

Traditionally, such decoupling would be achieved through polymorphism. In many cases, this translates into significant overhead, because concrete object types must be resolved at runtime. For example, if the traversal methods from Section 4.1 were defined through polymorphism, i.e., by accessing concrete traversal methods only through an abstract superclass, enumerating the elements of a sequence would require runtime resolution of the traversal type each time an element is visited. By using the C++ template techniques named *static polymorphism* [Coplien, 1995; Alexandrescu, 2001], such overhead is completely avoided. These techniques displace a significant part of the computing effort from runtime to compile-time, and allow function inlining, which can notably improve performance. This is similar to the approach used in the design of the containers from the C++ STL.

Besides, template techniques allow for more flexibility than polymorphism. For example, there are situations where the same operations must be carried on scalar merit values $\mathcal{D}_q(P_n)$ (for ordinary lattices) and on vectors of merit values $\mathcal{D}_q(P_{n_k})$ for $k = 1, \dots, m$ (for embedded lattices), for example, multiplication by a scalar value. This operation is supported with the same syntax for both types of merit values (scalar and vector). It follows that the same code that requires multiplying a (scalar or vector) merit value by a scalar can be used in both cases, despite the different types of the result of the operation (scalar or vector, respectively). This avoids large amounts of code duplication.

5 Usage

Lattice Builder can be used as a C++ software library from other programs, through its API, or as an executable program callable from its CLI. For detailed descriptions of the API and of the CLI options, the reader is referred to the user's guide included with the software package (see `cmdtut.html` for a tutorial on using the CLI). The CLI provides a simple means of invoking the most popular variants of the lattice construction algorithms covered in the previous sections. Lattice Builder has two software dependencies: the Boost [Boost.org, 2012] and FFTW [Frigo and Johnson, 2005] software libraries. Its code also uses some features of the C++11 standard.

On top of the CLI, there is also a graphical interface accessible through a web browser, which is more user friendly. It permits one to specify the various options and parameters available in the CLI, such as the dimension, number of points, figure of merit, weights, search method, etc., via menus. This interface is built with Python, so its use requires the availability of Python.

The set of tasks that Lattice Builder can accomplish is a Cartesian product in six dimensions, namely, the lattice type, the construction type, the accumulator type, the weight type, the figure of merit and the filter. The CLI reflects that.

It often happens that multiple generating vectors have approximately or exactly the same merit value. Since the precision of the computed merit values may depend on the platform on which the software is

executed, Lattice Builder may return different generating vectors on different platforms when making a search.

6 Examples of applications

In this section, we give a few examples of what can be done with Lattice Builder. L'Ecuyer and Munger [2012] provide other examples of applications where lattice rules built with (a preliminary version of) Lattice Builder and carefully selected weights produced RQMC estimators with smaller variance than rules with "general-purpose" parameters.

6.1 Importance of the weights

Our purpose here is to show that Lattice Builder can be used to obtain good lattice rules adapted to specific problems, and that this can yield a significant gain in simulation efficiency.

To see how the weights chosen to construct a lattice rule can have a direct impact on the RQMC variance, we consider as our integrand the worst-case function f_α^* from the space of functions $f : [0,1)^s \to \mathbb{R}$ with square variation $\mathcal{V}_{\mathfrak{u}}^2(f) \le v_{\mathfrak{u}}^2$ for each subset $\mathfrak{u}$ of coordinates, for given constants $v_{\mathfrak{u}}^2 \ge 0$. We obtain this worst-case function by replacing $\gamma_{\mathfrak{u}}$ by $v_{\mathfrak{u}}$ in (13):

$$f_\alpha^*(x_1,\ldots,x_s) = \sum_{\mathfrak{u}\subseteq\{1,\ldots,s\}} v_{\mathfrak{u}} \prod_{j\in\mathfrak{u}} \frac{(2\pi)^{\alpha/2}}{(\alpha/2)!} B_{\alpha/2}(x_j). \tag{33}$$

Since the RQMC variance of f_α^* is the value of $\mathcal{P}_\alpha$ with $\gamma_{\mathfrak{u}} = v_{\mathfrak{u}}$, there is no need to estimate the RQMC variance by simulation; it suffices to evaluate $\mathcal{P}_\alpha$, for any given lattice rule. In our experiments here, we fix the constants $v_{\mathfrak{u}}$ in (33) and we measure the increase of RQMC variance when we integrate (33) using lattice rules constructed via fast CBC with different weights $\gamma_{\mathfrak{u}} \ne v_{\mathfrak{u}}$ instead of with the ideal weights $\gamma_{\mathfrak{u}} = v_{\mathfrak{u}}$.

We start by considering a simple integrand f_α^* in dimension $s = 10$, such that all projections of the same order are equally important: we set $v_{\mathfrak{u}} = \Gamma^{|\mathfrak{u}|}$ for $|\mathfrak{u}| \le k$, and $v_{\mathfrak{u}} = 0$ otherwise, for a given integer $k > 0$ and a given positive constant $\Gamma \le 1$. Then, we select the weights in our criterion as $\gamma_{\mathfrak{u}} = \tilde{\Gamma}^{|\mathfrak{u}|}$ for $|\mathfrak{u}| \le \tilde{k}$, where the constants $\tilde{\Gamma}$ and $\tilde{k}$ may differ from Γ and k, respectively.

Table 1 shows the ratio between the RQMC variance with the lattice rule constructed with the (wrong) weights $\gamma_{\mathfrak{u}}$ as specified above, and the RQMC variance obtained with the ideal weights $v_{\mathfrak{u}}$, rounded to three significant digits, for different values of n. Six different cases are presented, labeled A1, A2, B1, B2, C1 and C2, and are explained below.

In case A1, we take $k = \tilde{k} = s = 10$, $\Gamma^2 = 0.1$, and $\tilde{\Gamma}^2 = 0.001$, so the weights of high-order projections are much too small, but still nonzero. There is an impact on the variance, but it is not so bad.

In A2, we take $k = \tilde{k} = s = 10$, $\Gamma^2 = 0.001$, and $\tilde{\Gamma}^2 = 0.1$, so the weights of high-order projections are much too large. We see that this has more impact on the RQMC variance. What happens is that the

Table 1: Ratio of RQMC variances for the estimator with modified weights to that with ideal weights.

n	A1	A2	B1	B2	C1	C2
2^8	1.11	1.21	1.13	4.08	3.82	6.80
2^9	1.21	1.10	1.42	10.5	2.93	7.25
2^{10}	1.36	1.38	2.04	4.64	2.86	5.94
2^{11}	1.24	1.43	2.40	6.18	2.15	5.14
2^{12}	1.42	1.66	3.79	13.2	2.47	5.94
2^{13}	1.30	2.38	5.51	9.09	2.66	5.97
2^{14}	1.51	2.54	**30.5**	8.66	**9.11**	**29.1**
2^{15}	1.46	1.93	25.6	**13.3**	3.52	9.71
2^{16}	**1.80**	**2.55**	3.13	12.9	2.73	10.2

search algorithm gives too much importance to the (several) higher-order projections, which results in a deterioration of the lower-order projections and has more impact on the variance.

In B1, we take $\Gamma^2 = \tilde{\Gamma}^2 = 0.1$, $k = 4$, and $\tilde{k} = 2$, so the search criterion is blind (gives no weight at all) to projections of order 3 and 4. This is case A1 carried to the extreme. Whereas all important projections had nonzero in case A1, here some important projections have no weight at all. As a result, the figure of merit cannot discriminate between lattices with good or bad projections of orders 3 or 4, and it is unpredictable whether the construction process will yield lattices with good or bad projections for these orders. Thus, the impact on the RQMC variance is unpredictable, significantly larger than in A1, and sometimes dramatic, as we can see for $n = 2^{14}$.

In B2, we take $\Gamma^2 = \tilde{\Gamma}^2 = 0.5$, $k = 2$, and $\tilde{k} = 4$. This gives weight to irrelevant projections of order 3 and 4. Here the degradation is even stronger for most values of n. The behavior is similar to case A2, but the observed degradation is much stronger here.

In case C1, we keep $\Gamma^2 = \tilde{\Gamma}^2 = 0.1$ and $k = \tilde{k} = 4$, but we increase the variation of f for a few projections by replacing $v_{\mathfrak{u}}^2$ with $v_{\mathfrak{u}}^2 + \tilde{v}_{\mathfrak{u}}^2$, where $\tilde{v}_{\mathfrak{u}}^2 = 1.0$ for $\mathfrak{u} = \{1,3\},\{3,5\},\{5,7\},\{7,9\}$, $\tilde{v}_{\mathfrak{u}}^2 = 0.5$ for $\mathfrak{u} = \{2,3,4\},\{4,5,6\},\{6,7,8\},\{8,9,10\}$, $\tilde{v}_{\mathfrak{u}}^2 = 0.25$ for $\mathfrak{u} = \{1,2,3,4\},\{4,5,6,7\},\{7,8,9,10\}$, and $\tilde{v}_{\mathfrak{u}} = 0$ for all other projections $\mathfrak{u}$. This means that some important projections are not given enough weight in the criterion relative to other projections. The impact on the variance is quite significant. Lattice Builder permits one to add (easily) extra weight to a few arbitrary projections that are more important. This example shows that it can really make a difference.

Case C2 is similar to C1, except that $v_{\mathfrak{u}}^2$ is replaced with only $\tilde{v}_{\mathfrak{u}}^2$, as defined above, instead of with $v_{\mathfrak{u}}^2 + \tilde{v}_{\mathfrak{u}}^2$, so that projections with $\tilde{v}_{\mathfrak{u}} = 0$ do not contribute at all to (33). That is, a lot of projections having nonzero weights are actually irrelevant. The degradation turns out to be much stronger than for C1. The explanation is similar to that in cases A2 and B2.

On our hardware configuration, it took Lattice Builder less than half a second to execute a fast CBC search for $n = 2^{16}$ with the ideal weights of case C1, i.e. a sum of order-dependent and projection-dependent weights. This is likely to be a negligible effort for many practical applications in simulation.

Note that in all these cases, the variance ratio tends to increase (non-monotonically) with n.

6.2 Competing projections

The results from Section 6.1 indicate that projections of different orders compete against each other in the sense that lattice rules with highly uniform projections of order k tend to have poor projections of order $\tilde{k} \neq k$ compared to other lattices rules with very good projections of order $\tilde{k}$. We examine this question from a different angle by considering the values of the spectral and $\mathcal{P}_2$ criteria across 1000 randomly selected lattice rules. For each lattice rule, we compare the merit value obtained by considering only projections of order 2 (the *order-2* case) to that obtained by considering only projections of order 3 (the *order-3* case). For the order-k case, with $k = 2$ or 3, we take $1/\Gamma_k^2$ equal to the number of projections of order k, and we set $\Gamma_{\tilde{k}} = 0$ for $\tilde{k} \neq k$. We consider lattice rules with $n = 2^{20}$, in $s = 3$ dimensions with $\Gamma_2^2 = 1/3$ and $\Gamma_3^2 = 1$, and in $s = 10$ dimensions with $\Gamma_2^2 = 1/45$ and $\Gamma_3^2 = 1/120$. The results are in Figure 2. In all cases, we see that a large portion of the selected lattices have both good order-2 and order-3 merit values. For the spectral criterion, the absence of data in the lower right corner of the plot suggests that having good projections of order 3 excludes having very bad projections of order 2. However, for the $\mathcal{P}_2$ criterion, low order-3 values are often associated with high order-2 values, and conversely. In particular, the lattice with the smallest order-3 $\mathcal{P}_2$ value (of 2.2×10^{-8}) with $s = 3$ has the third worst order-2 value (of 1.8×10^{-4}), several orders of magnitude above the best order-2 values (which are around 10^{-9}). We tried other values of n and of s and observed similar results.

6.3 Comparing construction methods

Here we compare the quality of lattice rules obtained with random CBC and standard CBC constructions. We select the weighted $\mathcal{P}_2$ figure of merit for product weights with $\gamma_j^2 = 0.1$ for all j, in $s = 10$ dimensions,

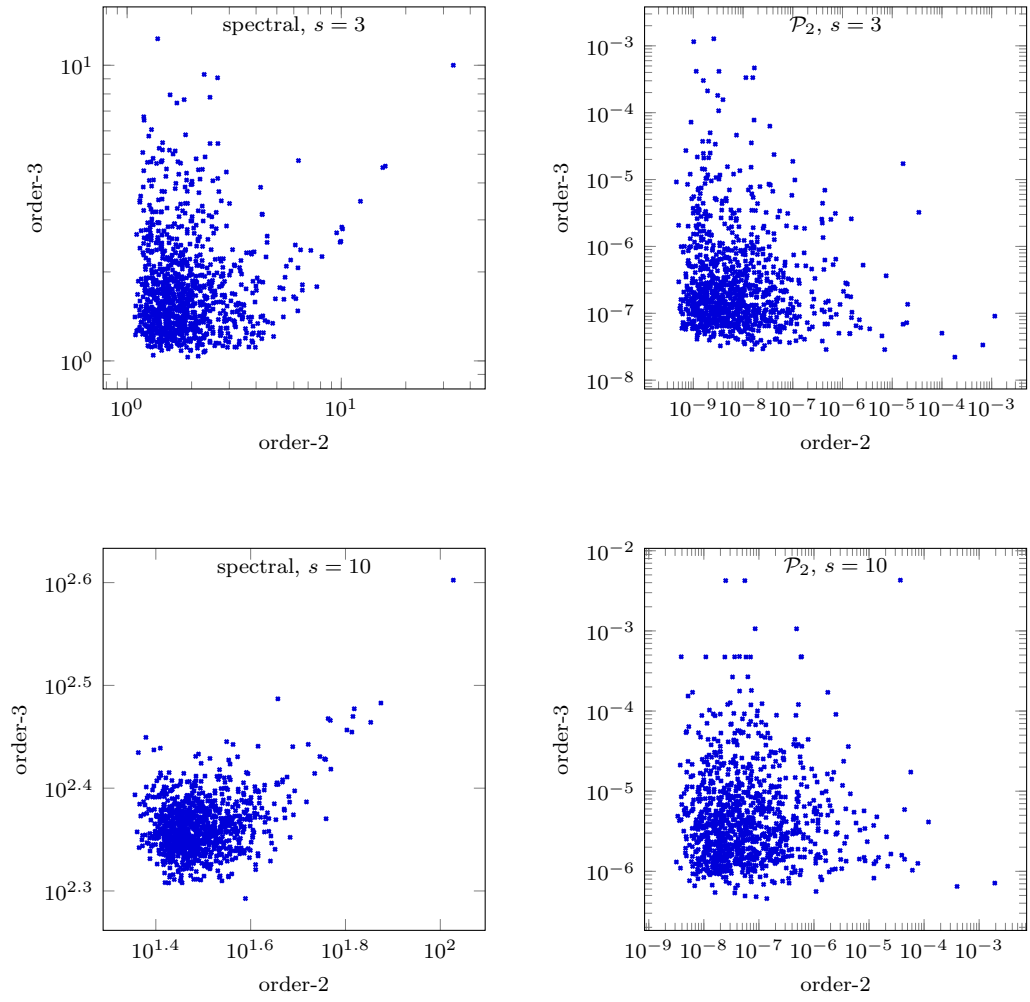

Figure 2: Order-2 versus order-3 spectral criterion (left) and $\mathcal{P}_2$ criterion (right), for 1000 random lattices with $n = 2^{20}$, and with $s = 3$ (top) and 10 (bottom). Note the different vertical and horizontal scales across the plots.

and we consider for n all powers of 2 from $2^5 = 32$ to $2^{14} = 16,384$. Let us define r_z as the smallest value of r for which the *expectation* of the merit value reached by random CBC is larger than that obtained with CBC by at most $z\,\%$. We estimate the expected merit value with the average of 300 independent realizations. We do estimate r_z with a bisection method even though the merit value of a lattice rule constructed by random CBC might be not always monotone in r To compute the merit value obtained with CBC, we use the fast CBC implementation. Figure 3 shows the estimated value of r_z obtained for each value of n, for $z = 5$, 10 and 20, in dimension $s = 10$. We see that r_z tends to become larger as n grows. This suggests that, when n is increased, r should also be increased for random CBC to yield lattice rules of consistent quality relative to CBC, but not as fast as n itself, at least for $z = 10$ and 20. Although not illustrated here, we also note that r can be kept smaller in larger dimensions.

6.4 Convergence of the quantiles

In this example, we use Lattice Builder to investigate the distribution of the values of the weighted $\mathcal{P}_\alpha$ figure of merit for $\alpha = 2$, with product weights with $\gamma_j^2 = 0.3$ for all j, for all generating vectors $\mathbf{a} \in \{1\} \times U_n^{s-1}$. We considered for n all powers of 2 from $2^6 = 64$ to $2^{19} = 524288$ for $s = 2$, and up to $n = 2^{15} = 32768$ for $s = 3$. There are $\varphi(n)^{s-1} = (n/2)^{s-1}$ different lattices to examine in each case. To do that, we wrote a program that uses the API of Lattice Builder (the source code is available with the software package). The program builds,

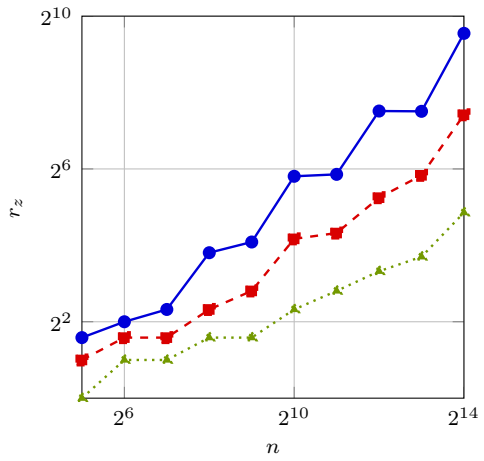

Figure 3: r_z as a function of n for $z = 5$ (—●—), $z = 10$ (- ■ -) and $z = 20$ (···▲···).

for each value of n considered, the cumulative distribution function (cdf) of the merit values, from which we extract the quantiles of the empirical distribution. Figure 4 displays a few quantiles as a function of n, as well as the average merit value. For $s = 2$, a linear regression on the logarithm of the merit value as a function of $\log n$ for $2^{12} \le n \le 2^{19}$ reveals that the best merit value decreases approximately as $n^{-1.92}$, the 10 % and 90 % quantiles decrease as $n^{-1.87}$ and $n^{-1.77}$ respectively (empirically), whereas the mean decreases as n^{-1}. For $s = 3$, by performing again a regression on the values of n such that $2^{10} \le n \le 2^{15}$, we found that the

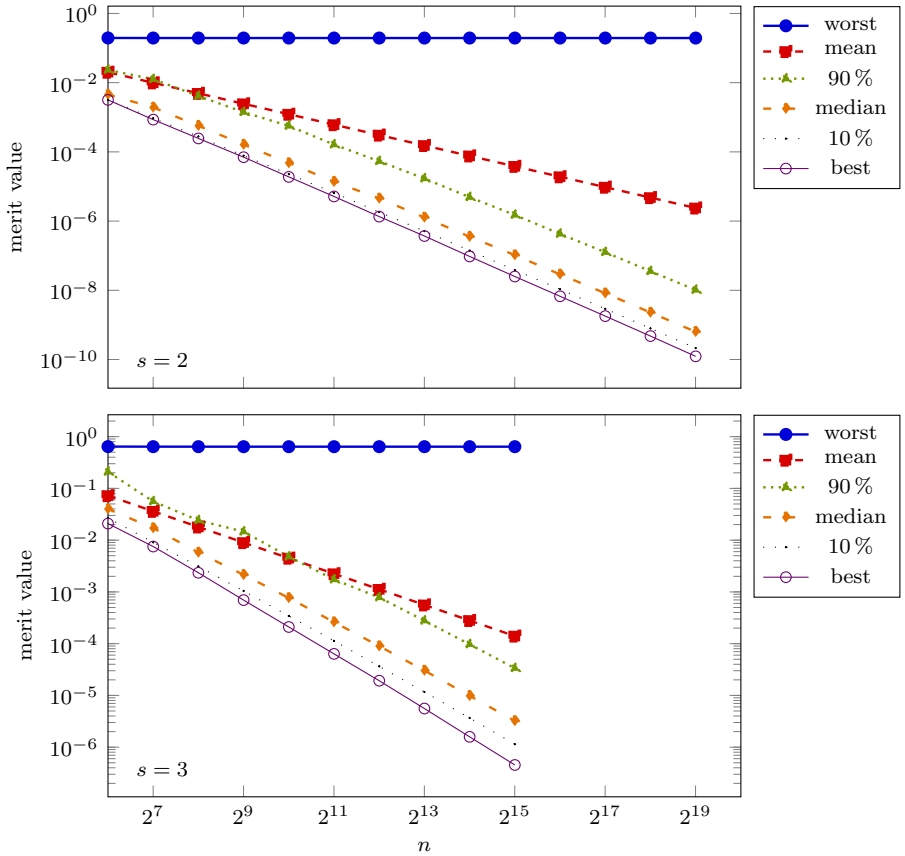

Figure 4: Convergence of the quantiles for all lattices in $\{1\} \times U_n^{s-1}$ for $s = 2$ and $s = 3$.

mean decreases at the same rate as for $s = 2$, while the best value and the 10 % and 90 % quantiles decrease (empirically) as $n^{-1.76}$, $n^{-1.64}$, and $n^{-1.39}$, respectively. The worst merit value stabilizes at around 0.1948 for $s = 2$ and around 0.6393 for $s = 3$. Interestingly, for $s = 2$ (and perhaps also for $s = 3$, but this is less obvious) the 90 % quantile appears to decrease at a slower rate to the left of $n = 2^{10} = 1024$ than to the right. These results indicate that the fraction of lattice rules with bad merit values decreases with n.

How well do CBC and random CBC perform in terms of the quantiles? This is what Figure 5 shows by comparing the merit values obtained with CBC and with random CBC with $r = 10$ and $r \approx \log n$ to those of the best merit value and of the 10 % quantile. In every case, the CBC curve remains very close to the best-value curve. On the other hand, both random-CBC curves are very similar and close to the 10 % quantile.

Finally, we note that for larger values of s, the results could be significantly different.

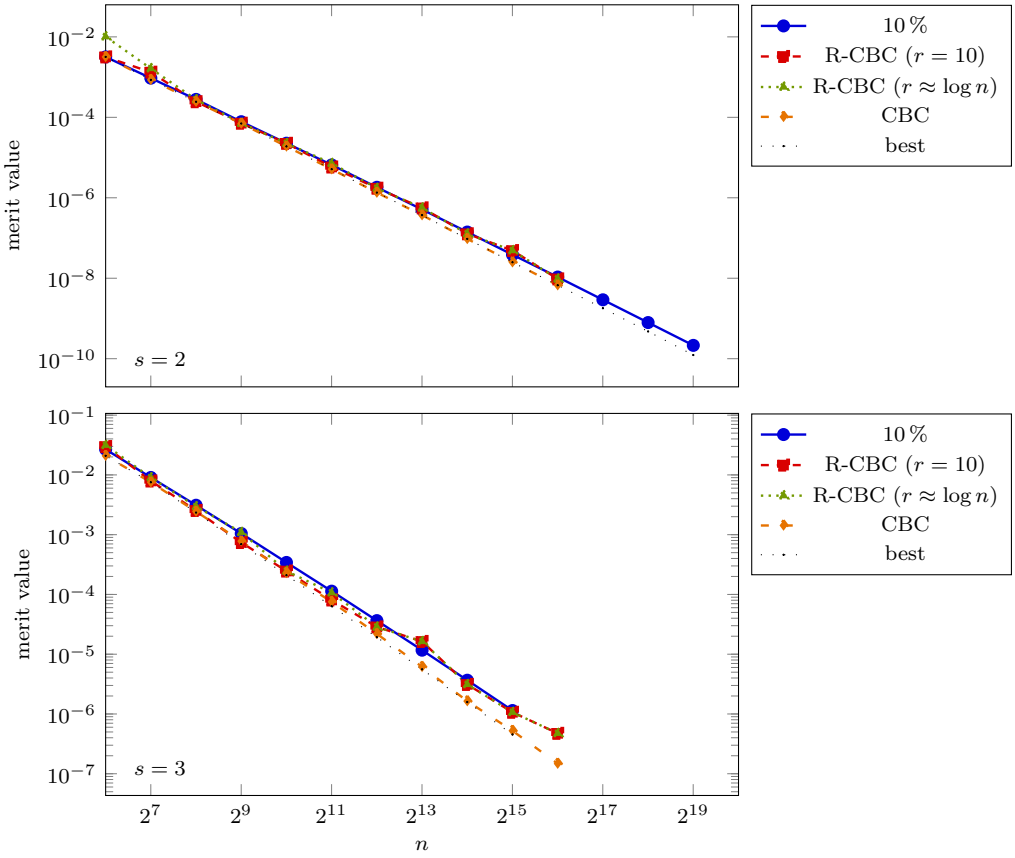

Figure 5: Convergence of CBC and random CBC for $s = 2$ and $s = 3$, compared to the quantiles for all lattices in $\{1\} \times U_n^{s-1}$.

6.5 Comparing $\mathcal{P}_\alpha$ and $\mathcal{R}_\alpha$

Here we examine empirically the distribution of the relative difference between $\mathcal{P}_\alpha$ and $\mathcal{R}_\alpha$ for random lattices, for a given α. Using Lattice Builder, we evaluated $\mathcal{P}_2$ and $\mathcal{R}_2$ for 1000 lattice point sets with $n = 2^{12}$, with randomly chosen generating vectors $\mathbf{a}$, in dimensions $s = 5$ and 10. We used geometric order-dependent weights $\gamma_\mathfrak{u} = \gamma^{|\mathfrak{u}|}$ with $\gamma^2 = 0.7$. We see from Figure 6 that the relative distance between $\mathcal{P}_2$ and $\mathcal{R}_2$ remains below 1 % for good lattices in both cases. For bad lattices (those with higher values of $\mathcal{P}_2$ and $\mathcal{R}_2$), the relative distance is even shorter. We tried with other values of s and observed a continuous deformation of the curve consistent with the two cases illustrated in Figure 6.

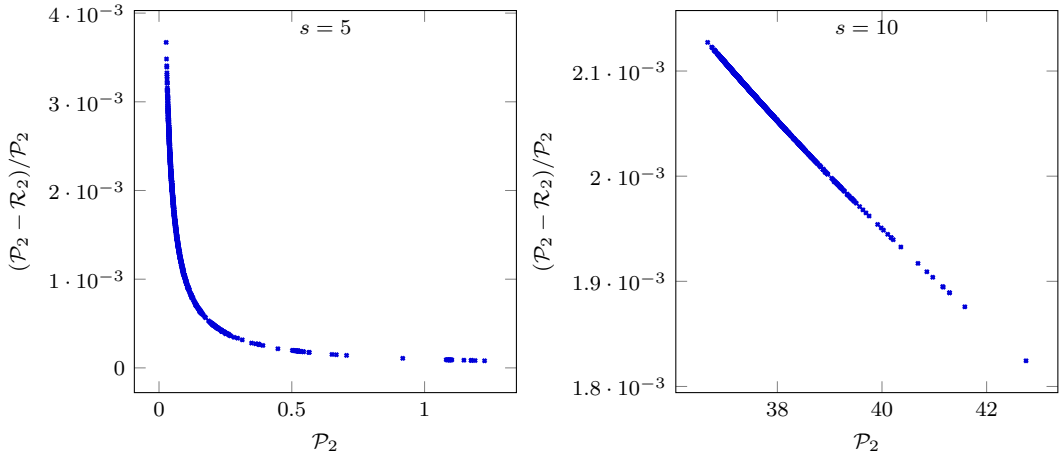

Figure 6: Distance from $\mathcal{R}_2$ to $\mathcal{P}_2$ in units of $\mathcal{P}_2$, in dimension $s = 5$ (left) and 10 (right), for 1000 random lattices with $n = 2^{12}$. Notice the different vertical and horizontal scales.

In Figure 7, we plot, as a function of the number of points, the relative distance from $\mathcal{R}_2$ to $\mathcal{P}_2$, for the lattice obtained using CBC search with the $\mathcal{P}_2$ criterion and product weights such that $\gamma_j^2 = j^{-1}$ for $j = 1, \ldots, s$, in dimensions $s = 2$, 5 and 10. The distance falls below 1 % for $n \approx 2^{20}$ for $s = 2$, and for $n \approx 2^6$ for $s = 5$ and 10. We also show (with gray dashed curves) the value of

$$B = \sum_{\varnothing \neq \mathfrak{u} \subseteq \{1,\ldots,s\}} \gamma_{\mathfrak{u}}^{2\alpha} \, B(\mathfrak{u})$$

with $B(\mathfrak{u})$ as in the bound derived by Hickernell and Niederreiter [2003], Lemma 4, under the form $\mathcal{P}_\alpha(P_n(\mathfrak{u})) < \mathcal{R}_\alpha(P_n(\mathfrak{u})) + B(\mathfrak{u})$. (These authors consider possible weights inside of $B(\mathfrak{u})$, but in their equation (13), they set all these weights to 1, as we do here). In all cases, the curves for B land orders of magnitude above those for the actual distance, but exhibit comparable decrease rates. We repeated the experiments with $\alpha = 4$ (not shown here) and observed that the relative distance from $\mathcal{R}_4$ to $\mathcal{P}_4$ decreased much more rapidly with n than with $\alpha = 2$.

For $s = 5$, we also computed the values of R_α for $\alpha = 1$ and 1.8 and compared these against those of $\mathcal{P}_2$ in Figure 8. For $\alpha = 1.8$, the values of $\mathcal{R}_\alpha$ remain well correlated with those of $\mathcal{P}_2$, with comparable orders of

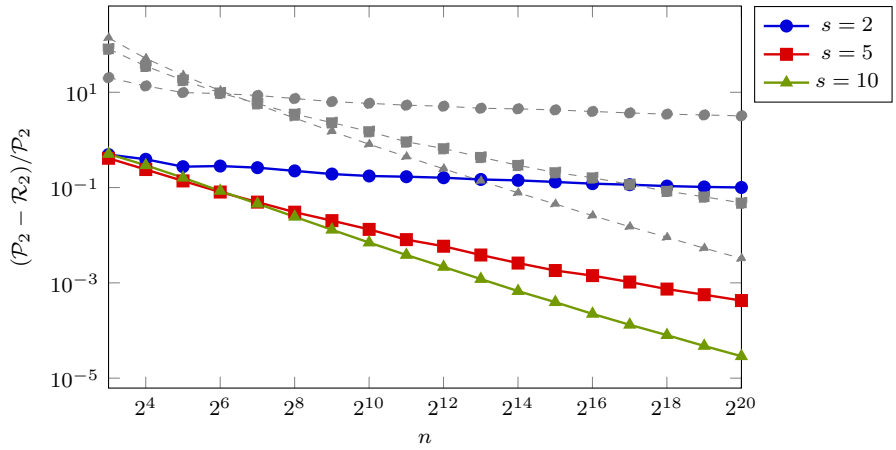

Figure 7: Distance from $\mathcal{R}_2$ to $\mathcal{P}_2$, in units of $\mathcal{P}_2$, for the lattice obtained using CBC search with the $\mathcal{P}_2$ criterion and product weights such that $\gamma_j^2 = j^{-1}$ for $j = 1, \ldots, s$, in dimensions $s = 2$, 5 and 10. The gray dashed lines correspond to the value of B as defined in the text.

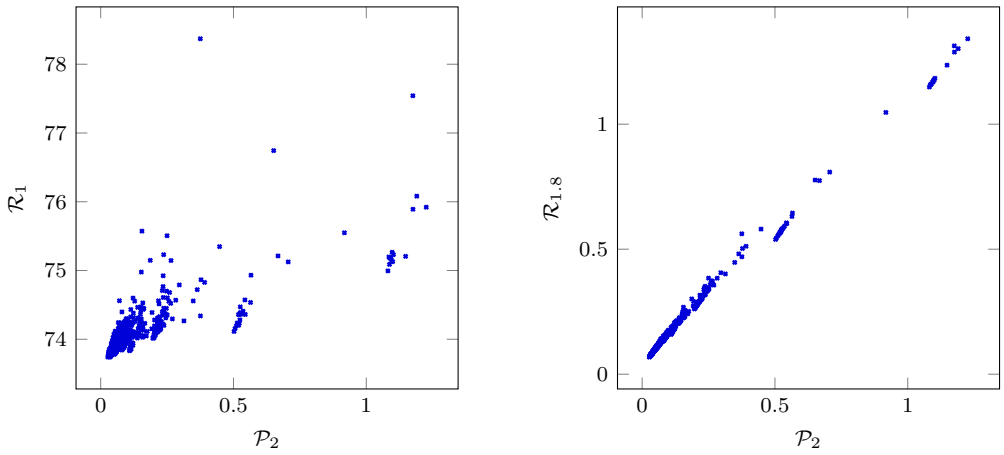

Figure 8: Values of $\mathcal{R}_\alpha$ against those of $\mathcal{P}_2$ of 1000 random lattices in dimension $s = 5$ with $n = 2^{12}$, for $\alpha = 1$ (left) and 1.8 (right). Notice the different vertical scales.

magnitude. For $\alpha = 1$ the values of $\mathcal{R}_\alpha$ are relatively much larger than those of $\mathcal{P}_2$ and not much correlated, although for good lattices, they seem more correlated. We also tried with other values of $1 \le \alpha \le 2$; the results looked like an interpolation between the two panels in Figure 8.

References

Alexandrescu, A. 2001. Modern C++ design: generic programming and design patterns applied. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Boost.org. 2012. Boost C++ libraries. http://www.boost.org.

Burkardt, J. 2012. LATTICE_RULE. http://people.sc.fsu.edu/~jburkardt/m_src/lattice_rule/lattice_rule.html.

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

Cools, R., Kuo, F. Y., and Nuyens, D. 2006. Constructing embedded lattice rules for multivariate integration. SIAM Journal on Scientific Computing 28, 16, 2162–2188.

Coplien, J. 1995. Curiously recurring template pattern. C++ Report, 24–27.

Cranley, R. and Patterson, T. N. L. 1976. Randomization of number theoretic methods for multiple integration. SIAM Journal on Numerical Analysis 13, 6, 904–914.

Dick, J. and Pillichshammer, F. 2010. Digital Nets and Sequences: Discrepancy Theory and Quasi-Monte Carlo Integration. Cambridge University Press, Cambridge, U.K.

Dick, J., Pillichshammer, F., and Waterhouse, B. J. 2008. The construction of good extensible rank-1 lattices. Mathematics of Computation 77, 264, 2345–2373.

Dick, J., Sloan, I. H., Wang, X., and Woźniakowski, H. 2004. Liberating the weights. Journal of Complexity 20, 5, 593–623.

Dick, J., Sloan, I. H., Wang, X., and Woźniakowski, H. 2006. Good lattice rules in weighted Korobov spaces with general weights. Numerische Mathematik 103, 63–97.

Efron, B. and Stein, C. 1981. The jackknife estimator of variance. Annals of Statistics 9, 586–596.

Frigo, M. and Johnson, S. G. 2005. The design and implementation of FFTW3. Proceedings of the IEEE 93, 2, 216–231. Special issue on "Program Generation, Optimization, and Platform Adaptation".

Hickernell, F. J. 1998a. A generalized discrepancy and quadrature error bound. Mathematics of Computation 67, 221, 299–322.

Hickernell, F. J. 1998b. Lattice rules: How well do they measure up? In Random and Quasi-Random Point Sets, P. Hellekalek and G. Larcher, Eds. Lecture Notes in Statistics Series, vol. 138. Springer-Verlag, New York, 109–166.

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

Hickernell, F. J. and Niederreiter, H. 2003. The existence of good extensible rank-1 lattices. Journal of Complexity 19, 3, 286–300.

Knuth, D. E. 1998. The Art of Computer Programming, Volume 2: Seminumerical Algorithms Third Ed. Addison-Wesley, Reading, MA.

Kuo, F. 2012. Lattice rule generating vectors. http://web.maths.unsw.edu.au/~fkuo/lattice/index.html.

Kuo, F. Y. and Joe, S. 2002. Component-by-component construction of good lattice rules with a composite number of points. Journal of Complexity 18, 4, 943–976.

Kuo, F. Y., Schwab, C., and Sloan, I. H. 2011. Quasi-Monte Carlo methods for high dimensional integration: the standard (weighted Hilbert space) setting and beyond. The ANZIAM Journal 53, 1–37.

Kuo, F. Y., Schwab, C., and Sloan, I. H. 2012. Quasi-Monte Carlo finite element methods for a class of elliptic partial differential equations with random coefficients. SIAM Journal on Numerical Analysis 50, 3351–3374.

Kuo, F. Y., Sloan, I. H., and Woźniakowski, H. 2008. Lattice rule algorithms for multivariate approximation in the average case setting. Journal of Complexity 24, 283–323.

Kuo, F. Y., Wasilkowski, G. W., and Waterhouse, B. J. 2006. Randomly shifted lattice rules for unbounded integrands. Journal of Complexity 22, 5, 630–651.

L'Ecuyer, P. 1999a. Tables of linear congruential generators of different sizes and good lattice structure. Mathematics of Computation 68, 225, 249–260.

L'Ecuyer, P. 1999b. Tables of maximally equidistributed combined LFSR generators. Mathematics of Computation 68, 225, 261–269.

L'Ecuyer, P. 2008. SSJ: A Java Library for Stochastic Simulation. Software user's guide, available at http://www.iro.umontreal.ca/~lecuyer.

L'Ecuyer, P. 2009. Quasi-Monte Carlo methods with applications in finance. Finance and Stochastics 13, 3, 307–349.

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

L'Ecuyer, P. and Lemieux, C. 2000. Variance reduction via lattice rules. Management Science 46, 9, 1214–1235.

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

L'Ecuyer, P., Munger, D., and Tuffin, B. 2010. On the distribution of integration error by randomly-shifted lattice rules. Electronic Journal of Statistics 4, 950–993.

L'Ecuyer, P. and Owen, A. B., Eds. 2009. Monte Carlo and Quasi-Monte Carlo Methods 2008. Springer-Verlag, Berlin.

Lemieux, C. 2009. Monte Carlo and Quasi-Monte Carlo Sampling. Springer-Verlag, New York, NY.

Lemieux, C. 2012. RandQMC library. http://www.math.uwaterloo.ca/~clemieux/randqmc.html.

Lischner, R. 2009. Exploring C++: The Programmer's Introduction to C++. Apress, Books24x7. Web.

Liu, R. and Owen, A. B. 2006. Estimating mean dimensionality of analysis of variance decompositions. Journal of the American Statistical Association *101,* 474, 712–721.

Maisonneuve, D. 1972. Recherche et utilisation des "bons treillis", programmation et résultats numériques. In Applications of Number Theory to Numerical Analysis, S. K. Zaremba, Ed. Academic Press, New York, NY, 121–201.

Niederreiter, H. 1992a. New methods for pseudorandom number and pseudorandom vector generation. In Proceedings of the 1992 Winter Simulation Conference. IEEE Press, 264–269.

Niederreiter, H. 1992b. Random Number Generation and Quasi-Monte Carlo Methods. SIAM CBMS-NSF Regional Conference Series in Applied Mathematics Series, vol. 63. SIAM, Philadelphia, PA.

Nuyens, D. 2012. Fast component-by-component constructions. http://people.cs.kuleuven.be/~dirk.nuyens/fast-cbc/.

Nuyens, D. 2014. The construction of good lattice rules and polynomial lattice rules. In Radon Series on Computational and Applied Mathematics, P. Kritzer, H. Niederreiter, F. Pillichshammer, and A. Winterhof, Eds. De Gruyter. to appear.

Nuyens, D. and Cools, R. 2006a. Fast algorithms for component-by-component construction of rank-1 lattice rules in shift-invariant reproducing kernel Hilbert spaces. Mathematics of Computation 75, 903–920.

Nuyens, D. and Cools, R. 2006b. Fast component-by-component construction, a reprise for different kernels. In Monte Carlo and Quasi-Monte Carlo Methods 2004, H. Niederreiter and D. Talay, Eds. 373–387.

Nuyens, D. and Cools, R. 2006c. Fast component-by-component construction of rank-1 lattice rules with a non-prime number of points. Journal of Complexity 22, 4–28.

Owen, A. B. 1998. Latin supercube sampling for very high-dimensional simulations. ACM Transactions on Modeling and Computer Simulation *8,* 1, 71–102.

Sinescu, V. and L'Ecuyer, P. 2009. On the behavior of weighted star discrepancy bounds for shifted lattice rules. In Monte Carlo and Quasi-Monte Carlo Methods 2008, P. L'Ecuyer and A. B. Owen, Eds. Springer-Verlag, Berlin, 603–616.

Sinescu, V. and L'Ecuyer, P. 2012. Variance bounds and existence results for randomly shifted lattice rules. Journal of Computational and Applied Mathematics 236, 3296–3307.

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

Sloan, I. H. and Woźniakowski, H. 1998. When are quasi-Monte Carlo algorithms efficient for high-dimensional integrals. Journal of Complexity 14, 1–33.

Sobol', I. M. 2001. Global sensitivity indices for nonlinear mathematical models and their Monte Carlo estimates. Mathematics and Computers in Simulation 55, 271–280.

Wang, X. 2007. Constructing robust good lattice rules for computational finance. SIAM Journal on Scientific Computing 29, 2, 598–621.

Wang, X. and Sloan, I. H. 2006. Efficient weighted lattice rules with applications to finance. SIAM Journal on Scientific Computing 28, 2, 728–750.

Woźniakowski, H. and Plaskota, L., Eds. 2012. Monte Carlo and Quasi-Monte Carlo Methods 2010. Springer-Verlag, Berlin.