Version: January 13, 2004

RandQMC

User's guide

A package for randomized quasi-Monte Carlo methods in C

Christiane Lemieux
Mikolaj Cieslak
Kristopher Luttmer

University of Calgary

Abstract. This package contains implementations for many quasi-Monte Carlo methods and their associated randomizations. It is designed so that the same program shell can be used for all methods, with only a different input file in each case. Some of the methods included in the package can deal with an infinite (random) dimension, and most of the others have a limit of 360 on the dimension. A program that correctly generates the input file is provided with the package, as well as some basic statistical tools.

Contents

1 Introduction

With this package, our goal is to provide a comprehensive toolbox for applying quasi-Monte Carlo methods on multidimensional integration problems. Our main concern while designing this package was to build something that would be easy to use, easy to make available on public domain, and easy to update. The C language was chosen because of its wide use and accessibility.

There already exist publicly available code – see `www.mcqmc.org` – and commercial packages – e.g., *QR Streams* at `www.mathdirect.com/products/qrn/`, the *FinDer* software [36] at `www.cs.columbia.edu/~ap/html/finder.html` – implementing a large number of quasi-Monte Carlo methods. Also, a C++ library called *libseq* has been developed by Friedel and Keller [9]. It uses efficient algorithms to generate scrambled digital sequences, and other techniques such as Latin Hypercube and Supercube Sampling [28, 34]. It can be found at `www.multires.caltech.edu/software/libseq/index.html`. Recently, the same authors implemented another package called *SamplePack* that again has very fast implementations for nets in base 2 and various randomization methods, as well as tools to analyze the quality of point sets. It is available at `http://www.uni-kl.de/AG-Heinrich/DiscrePack.html`. The distinctive features of our package are that it includes a different choice of quasi-Monte Carlo methods and randomizations, it can deal with infinite dimension, and it is designed so that the same program shell can be used for all methods, with only a different input file in each case. This is particularly convenient when one wants to compare several methods/randomizations without having to change the main program every time. The files required to use our package are available at `http://www.math.ucalgary.ca/~lemieux/RandQMC.html`.

The general problem for which this package has been designed can be described as follows. The goal is to estimate the integral of a square-integrable function $f : [0,1)^s \to \mathbb{R}$, given by

$$\mu = \int_{[0,1)^s} f(\mathbf{u}) d\mathbf{u}.$$

Here, s is the dimension of the problem, and it represents the number of variables on which f depends. The case where f depends on an unbounded number of variables can be handled by our package provided the user chooses Korobov rules [15], Korobov polynomial rules [21], or the Monte Carlo method. All the methods provided by our package amount to choose a point set P_n and estimate μ by

$$Q_n = \frac{1}{n} \sum_{\mathbf{u}_i \in P_n} f(\mathbf{u}_i). \tag{1}$$

The Monte Carlo method amounts to take P_n as a set of n i.i.d. uniform points over $[0,1)^s$. The estimator Q_n is then unbiased since each $\mathbf{u}_i$ is uniformly distributed over $[0,1)^s$. A (non-randomized) quasi-Monte Carlo method amounts to choose a deterministic point set P_n that is more "regularly distributed" over $[0,1)^s$ than a set of n i.i.d. random points. There are many different ways of measuring how regularly distributed is a point set, which more or less all rely on a particular measure of *discrepancy* between the empirical distribution

induced by P_n and the uniform distribution over $[0,1)^s$. Examples of such measures will be given as we describe the methods provided in this package.

Randomized quasi-Monte Carlo methods consists in choosing a quasi-Monte Carlo point set, and then an appropriate randomization method. All the randomization methods provided in this package work as follows: a random vector Δ has to be generated (e.g., a vector in $[0,1)^s$, a vector of random bits, etc.), and then a certain function χ is applied to each point $\mathbf{u}_i$ in P_n using the random vector Δ so that $\chi(\Delta, \mathbf{u}_i)$ has a uniform distribution over $[0,1)^s$. In this way, if we denote by $\tilde{P}_n$ the randomized point set and replace P_n by $\tilde{P}_n$ in (1), the estimator Q_n is unbiased and with an appropriate choice of P_n and χ, it can have a lower variance than the Monte Carlo estimator.

In order to estimate the variance of a randomized quasi-Monte Carlo estimator, one needs to generate a certain number m of i.i.d. copies of Q_n. Hence a typical program requires:

(1) a function that corresponds to the evaluation of f at a given point $\mathbf{u}$;

(2) a mechanism by which the points $\mathbf{u}_i$ of the chosen QMC method are generated;

(3) a mechanism by which the randomization χ can be applied;

(4) a way of resetting these mechanisms so that a new copy of Q_n can be generated if desired;

(5) a mechanism that can be used inside the evaluation of f to generate additional coordinates of $\mathbf{u}_i$ when f depends on an unbounded number of variables;

(6) some statistical tools that can accumulate the observations of Q_n and compute an estimator for μ and $\mathrm{Var}(Q_n)$.

The first task is left to the user, and we provide some examples in Appendix A. We explain in Section 3 what functions are provided by our package to perform tasks (2)-(4), and illustrate with an example how they should be called in the user's program. As explained in Section 3, these functions are all called using the same names, and the information describing which method should be used is stored in an input file whose structure is explained in Section 4. The program `makeinput` described in Section 5 can create this input file. Section 6 gives a detailed description of how the procedures related to task (2) work specifically for each method, and more details on task (3) are provided in Section 7. The procedures required to perform the fifth task are introduced in Section 8. The last task listed above is handled by the `StatsQMC` library, which is presented in Section 9.

In the next section, we describe the methods provided in this package and the randomizations that can be used with them, along with references that contain more information.

2 Quasi-Monte Carlo Methods and Their Randomizations

We first describe how each method implemented in this package constructs the point set P_n, then explain how the available randomizations work, and which methods can they be used with.

2.1 Monte Carlo

As mentioned in the introduction, in this case P_n is obtained by taking n independent random points uniformly distributed over $[0,1)^s$. In practice, each point $\mathbf{u}_i$ is obtained by calling s times a pseudorandom number generator. In our package, the generator used for this purpose is the combined multiple recursive generator `MRG32k3a` described in [17], and we use the implementation given in [23].

2.2 Quasi-Monte Carlo methods

We now describe the QMC methods provided in this package. In Subsection 2.3, we explain how to speed up the generation of points from a digital net using a Gray code.

2.2.1 Korobov method

Here, the construction of P_n requires one parameter a called the *generator*, which is an integer between 1 and $n-1$. We then have

$$P_n = \left\{ \mathbf{u}_i = \frac{i}{n}(1, a, \ldots, a^{s-1}) \bmod 1, i = 0, \ldots, n-1 \right\}.$$

In the case where n is prime and a is a primitive element modulo n, P_n can be generated by running the following linear recurrence

$$x_i = a x_{i-1} \bmod n,$$

starting at any $x_0 \neq 0$, and then P_n can be obtained as follows:

$$P_n = \left\{ \frac{1}{n}(x_0, \ldots, x_{s-1}), x_0 = 0, \ldots, n-1 \right\}.$$

The choice of the generator a is important for the success of this method, and therefore it should be carefully chosen. In our implementation, we restrict the choice of n to be a prime or a power of two; when n is a power of two, a has to be an odd number. These conditions

guarantee that P_n is *dimension-stationary* and *fully projection-regular* [20], which are two important properties of the projections $P_n(I)$ of P_n, where for $I = \{i_1, \ldots, i_t\} \subseteq \{1, \ldots, s\}$,

$$P_n(I) = \{(u_{i,i_1}, \ldots, u_{i,i_t}), i = 0, \ldots, n-1\}.$$

The first property means that if $J = \{i_1 + \Delta, \ldots, i_t + \Delta\}$, with $0 \leq \Delta \leq s - i_t$, then $P_n(I) = P_n(J)$. For example, if $I = \{1, 3\}$, then $P_n(I) = P_n(J)$ for all subsets J of the form $\{1 + \Delta, 3 + \Delta\}$, for $0 \leq \Delta \leq s - 3$. The second property means that all projections contain n distinct points. The Korobov method is the only type of integration lattice that has these two properties. For this reason, we do not provide – at least not for now – more general implementations of integration lattices in our package.

If the user does not know what value of a should be chosen, the package offers some predetermined values for certain values of n (n = largest prime smaller or equal to 2^k, for $k = 10, \ldots, 17$, and $n = 2^k$ for $k = 7, \ldots, 20$), and these values have been chosen according to the criterion $M_{32,24,12,8}$ described in [20]. This criterion is based on the *spectral test*, which is often used to assess the quality of pseudorandom number generators [11, 14, 18]. It considers the lattice structure of P_n, and makes sure there is not too much spacing between the family of equidistant parallel hyperplanes on which the points of P_n lie. More precisely, the euclidean distance between the family of hyperplanes that are the farthest apart is computed. This distance turns out to be equal to the inverse of the length of the shortest vector in the dual lattice to P_n, which is defined as

$$L^* = \{\mathbf{h} \in \mathbb{Z}^s : \mathbf{h} \cdot \mathbf{u}_i \in \mathbb{Z}, \text{ for all } \mathbf{u}_i \in P_n\}.$$

One can compare this length, defined by

$$l_s = \min_{\mathbf{0} \neq \mathbf{h} \in L^*} \|\mathbf{h}\|_2,$$

with the best (largest) possible one l_s^* for any lattice in s dimensions (the exact value of l_s^* is known for $s \leq 8$, and bounds are available for $s > 8$ [5, 18]), i.e., the ratio

$$l_s / l_s^* \tag{2}$$

is computed, and the goal is to find a point set P_n for which (2) is as close to 1 as possible. This can be applied not only to P_n, but to any projection $P_n(I)$; for example, the criterion $M_{32,24,12,8}$ considers all projections $P_n(I)$ with I of the form $I = \{1, \ldots, t\}$, and $t \leq 32$; then pairs $I = \{1, i\}$ with $1 < i \leq 24$; triplets $I = \{1, i, j\}$ with $1 < i < j \leq 12$, and quadruplets $I = \{1, i, j, k\}$ with $1 < i < j < k \leq 8$. There is no loss of generality in assuming that the smallest index in I is 1 when P_n is dimension-stationary.

For more details on the Korobov method and integration lattices in general, we refer the reader to [15, 20, 39] and the references cited there.

2.2.2 Sobol' sequence

Sobol's method [40] was the first construction in the family of digital nets [31]. Each one-dimensional projection $P_n(\{j\})$ of P_n requires two things: first, a primitive polynomial $f_j(z)$ over $\mathbb{F}_2$, the finite field with two elements; second, an integer m_j to initialize a recurrence based on $f_j(z)$ that generates the *direction numbers* defining $P_n(\{j\})$. The method specifies that the polynomial $f_j(z)$ should be the j^{th} one in the list of primitive polynomials sorted by increasing degree (within each degree, Sobol' specifies a certain order that is given in the code of Bratley and Fox [2] for $j \leq 40$, and we used for $j > 40$ the order given in the list of primitive polynomials that can be found at Florent Chabaud's website `fchabaud.free.fr`).

Assume $f_j(z) = z^q + a_{j,1}z^{q-1} + \ldots + a_{j,q}$, where $a_{j,l} \in \mathbb{F}_2$ for each j, l. The direction numbers $v_{j,1}, v_{j,2}, \ldots$ are rationals of the form

$$v_{j,k} = \frac{m_{j,k}}{2^k} = \sum_{l=1}^{k} v_{j,k,l} 2^{-l},$$

where $m_{j,k}$ is an odd integer smaller than 2^k. One has to choose the first q values $v_{j,1}, \ldots, v_{j,q}$, (or equivalently, $m_{j,1}, \ldots, m_{j,q}$), and the following ones are obtained through the recurrence

$$v_{j,k} = a_{j,1}v_{j,k-1} \oplus \ldots \oplus a_{j,q-1}v_{j,k-q+1} \oplus v_{j,k-q} \oplus (v_{j,k-q}/2^q),$$

where $\oplus$ denotes a bit-by-bit exclusive-or operation, and $v_{j,k-q}/2^q$ means that the binary expansion of $v_{j,k-q}$ is shifted by q positions to the right. These direction numbers are then used to define $P_n(\{j\}) = \{u_{i,j}, i = 0, \ldots, n-1\}$ as follows:

$$u_{i,j} = i_0 v_{j,1} \oplus i_1 v_{j,2} \oplus \ldots \oplus i_{d-1} v_{j,d}, \tag{3}$$

where $i_0, \ldots, i_{d-1}$ are the coefficients in the binary expansion of i, i.e.,

$$i = \sum_{l=0}^{d-1} i_l 2^l,$$

and d is such that $n = 2^d$. An alternative description that fits the general construction principles for digital nets [31, 45] is to consider the binary expansion of $u_{i,j}$, given by

$$u_{i,j} = \sum_{l=1}^{L} u_{i,j,l} 2^{-l}.$$

Then we have that (3) is equivalent to

$$\mathbf{C}^j \begin{pmatrix} i_0 \\ i_1 \\ \vdots \\ i_{d-1} \end{pmatrix} = \begin{pmatrix} u_{i,j,1} \\ u_{i,j,2} \\ \vdots \\ u_{i,j,L} \end{pmatrix}, \tag{4}$$

where $\mathbf{C}^j$ is a $L \times d$ matrix defined by

$$\mathbf{C}^j_{l,k} = v_{j,k,l},$$

and the operations in (4) are performed in $\mathbb{Z}_2$.

In the implementation of Bratley and Fox [2], the initial values of $m_{j,k}$ for $j \leq 40$ given in [42] are provided. For dimension $40 < j \leq 360$, we have searched for "good" initial values based on a criterion that was inspired by criteria used in [4, 26, 41]. Our criterion is based on the *resolution* of P_n, which is a measure that is often used to assess the quality of pseudorandom number generators based on linear recurrences modulo 2 [6, 16, 45, 48]. More generally, for a point set P_n whose construction is done over $\mathbb{F}_b$ and such that $n = b^d$, the resolution is the largest integer ℓ_s such that each cubic cell obtained by partitioning $[0,1)^s$ into $b^{s\ell_s}$ cubic cells of same shape and size contains $b^{d-s\ell_s}$ points of P_n. In the same way as the criterion $M_{t_1,\ldots,t_k}$ described for choosing Korobov point sets considers many projections of P_n, one can define a similar criterion $\Delta_{t_1,\ldots,t_k}$ that computes the resolution for many projections, and that outputs the largest difference obtained between the resolution of a projection $P_n(I)$ and the maximal one, given by $\lfloor d/|I| \rfloor$; see [26] for the details.

The initial values of $m_{j,k}$ provided in [42] are based on two properties of P_n: "Property A" amounts to make sure ℓ_s is equal to s when $n = 2^s$, and "Property A' " means that the resolution $\ell_s = 2s$ when $n = 2^{2s}$ [40, 41]. In [4], the authors choose initial values for $m_{j,k}$ by measuring the quality of (some of) the two-dimensional projections of P_n using a measure of discrepancy harder to compute than the resolution. In our package, we chose the initial values $m_{j,k}$ for dimension j as follows: we computed the resolution of $P_n(\{j-i, j\})$ for $i = 1, \ldots, 8$, and $n = 2^d$, where d is the degree of the primitive polynomial assigned to dimension j, compared it with the maximal resolution $\lfloor d/2 \rfloor$, and measured what was the largest difference for those eight projections. Note that this criterion is not equivalent to using $\Delta_{2,8}$ since P_n is not dimension-stationary. The set of initial values we chose was the one providing the smallest maximal difference. We performed a random search to find these sets of initial values since the search space would otherwise have been too large. The optimal values of $m_{j,k}$ obtained for $40 < j \leq 360$ are available from the authors.

Our implementation of Sobol' sequence is based on the Fortran code of Bratley and Fox available at `www.acm.org/calgo`.

2.2.3 Shift-net

This method was proposed in [38]. Like Sobol' sequence, it is a special case of a digital net in base 2, i.e., the point set P_n is obtained by choosing s matrices $\mathbf{C}^1, \ldots, \mathbf{C}^s$ with elements in $\mathbb{Z}_2$, and then each coordinate u_{ij} is obtained via (4), with $L = d$, when $n = 2^d$.

The idea of shift-nets is to define the matrices $\mathbf{C}^2, \ldots, \mathbf{C}^s$ as functions of $\mathbf{C}^1$ as follows: assume $d = s$ and let

$$\mathbf{C}^1 = (\, \mathbf{c}^1_1 \quad \mathbf{c}^1_2 \quad \ldots \quad \mathbf{c}^1_s \,),$$

i.e., $\mathbf{c}_j^1$ represents the j^{th} column of $\mathbf{C}^1$. For $2 \leq j \leq s$, let

$$\mathbf{C}^j = (\, \mathbf{c}_j^1 \quad \ldots \quad \mathbf{c}_s^1 \quad \mathbf{c}_1^1 \quad \ldots \quad \mathbf{c}_{j-1}^1 \,),$$

i.e., $\mathbf{C}^j$ is obtained by shifting the columns of $\mathbf{C}^1$ by $j-1$ positions to the left.

Descriptions of matrices $\mathbf{C}^1$ that optimize the quality parameter t for digital nets for $d = s \leq 39$ are listed in [38]. Using the so-called *propagation rules*, it is possible to use these parameters to define point sets with $d \neq s$, and we offer this possibility in our package. Details are given in Subsection 6.5.

The quality parameter t mentioned previously is often used to assess the quality of digital nets in arbitrary base b, and has lead to the notion of (t, d, s)-nets [31]. To describe it, it is convenient to introduce the notion of $(q_1, \ldots, q_s)$-*equidistribution* [26]. Let $q = q_1 + \ldots + q_s$ and consider the b^q boxes obtained by partitioning $[0,1)^s$ into b^{q_j} equal intervals along the j^{th} axis. If each of these b^q boxes contains b^{d-q} points of an arbitrary point set P_n, where $n = b^d$, P_n is said to be $(q_1, \ldots, q_s)$-equidistributed. The quality parameter t is the smallest integer such that P_n is $(q_1, \ldots, q_s)$-equidistributed whenever $q \leq d - t$, and P_n is then called a (t, d, s)-net. Obviously, the quality parameter t is meaningless if $d \leq t$. Hence, if one uses less than 2^{t+1} points, there is generally no guarantee on the quality of P_n. We provide in Section 6 known values of t for Sobol' sequence [32], shift-nets [38], and the so-called "Salzburg Tables" [37]. In the literature, references to (t, s)-*sequences* are often made. These sequences $\mathbf{u}_0, \mathbf{u}_1, \ldots$ are such that for any integer m, the point set P_n defined by

$$P_n = \{\mathbf{u}_{mb^d}, \ldots, \mathbf{u}_{(m+1)b^d-1}\}$$

is a (t, d, s)-net in base b.

2.2.4 Generalized Faure sequences

These methods are special cases of digital nets in base b, where b is (generally) the smallest prime larger or equal to the dimension s. The quality parameter t for these constructions is 0, provided n is of the form $n = \lambda b^d$, for some integers $\lambda, d \geq 1$. Assume that in (4), $i_0, \ldots, i_{d-1}$ and $u_{i,j,1}, u_{i,j,2}, \ldots$, now represent the coefficients of the expansion of i and $u_{i,j}$ in base b, respectively. Similarly, the matrices $\mathbf{C}^j$ now contain integers in $\mathbb{Z}_b$ (assuming b is prime, as is the case in our package), and they may contain more than d rows, as specified in [45]. For generalized Faure sequences, these matrices take the following form:

$$\mathbf{C}^j = \mathbf{A}_j (\mathbf{P}^T)^{j-1},$$

where $\mathbf{P}$ is the $L \times L$ Pascal's matrix (i.e., $\mathbf{P}_{i,j} = \binom{i-1}{j-1} \pmod b$), and $\mathbf{A}_j$ is an arbitrary $L \times L$ non-singular lower-triangular matrix. One can then generate up to $n = b^L$ points using these matrices: see [8] for more details. The Faure sequence is obtained by taking $\mathbf{A}_j$ as the identity matrix for all $j = 1, \ldots, s$. The Generalized Faure sequence of Tezuka and Tokuyama [47] amounts to take $\mathbf{A}_j = \mathbf{P}^{j-1}$. Recently, Faure suggested another form of Generalized Faure sequence that consists in taking $\mathbf{A}_j$ equal to the lower-triangular matrix with all nonzero entries equal to 1, for all j. All three definitions are implemented in our package. Our implementation uses the code given by Tezuka [45, Appendix A], which generates P_n (assuming the matrices $\mathbf{C}_j$ have been pre-calculated).

2.2.5 Halton sequence

This quasi-Monte Carlo method was proposed by Halton in 1960 [10]. It is not a digital net construction, but can be seen as one of their "ancestors". Let p_j denote the j^{th} prime number (e.g., $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, etc.). Expanding i in base p_j, we get

$$i = \sum_{l=0}^{d_j - 1} i_{j,l}\, p_j^l,$$

and for $n \leq p_s^{d_s}$, the set P_n is obtained by letting

$$u_{ij} = \sum_{l=1} i_{j,l-1}\, p_j^{-l}.$$

In our package, it has been implemented up to dimension 360.

2.2.6 Randomized Generalized Halton Sequences

This method can be viewed as a randomized version of the Halton sequence, as described in [43]. However, the randomization used is specifically designed for Halton sequences and therefore we did not include it with the more general randomization techniques discussed in Section 2.4. For the same reason, it has been implemented in our package as a distinct construction rather than as a randomization.

The idea is to generate the first point randomly and then, the next point is obtained by applying in each dimension j a p_j-adic Van Neumann-Kakutani transformation on the base p_j representation of the j^{th} coordinate of the previous point. The same procedure is applied to get each subsequent point. See [43] for more details.

2.2.7 Salzburg Tables

The method for which these tables provide parameters is a special case of a digital net in base b, where b is an arbitrary prime power. This special case is described in [31, Section 4.4], and parameters that optimize the quality criterion t can be found in [37]. This method also corresponds to a *polynomial lattice rule of rank 1*, as described in [26]. The point set P_n is determined by a polynomial $f(z) = z^d + a_1 z^{d-1} + \ldots + a_d$ of degree d over $\mathbb{F}_b$, and s polynomials $g_j(z)$ also in $\mathbb{F}_b[z]$, of degree less than d. For each dimension $j = 1, \ldots, s$, consider the formal Laurent series expansion

$$\frac{g_j(z)}{f(z)} = \sum_{l=1}^{\infty} w_l^{(j)} z^{-l}.$$

The coefficients $w_l^{(j)}$ satisfy

$$\mathbf{A} \begin{pmatrix} w_1^{(j)} \\ \vdots \\ w_d^{(j)} \end{pmatrix} = \begin{pmatrix} g_{j1} \\ \vdots \\ g_{jd} \end{pmatrix},$$

where $\mathbf{A}$ is such that

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & \dots & 0 \\ a_1 & 1 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ a_{d-1} & \dots & a_1 & 1 \end{pmatrix},$$

and the coefficients g_{jl} are such that $g_j(z) = \sum_{l=1}^{d} g_{jl} z^{d-l}$ (see, e.g., [26]). The matrices $\mathbf{C}^1, \dots, \mathbf{C}^s$ are then defined as

$$\mathbf{C}_{i,l}^j = w_{i+l-1}^{(j)}.$$

For now, our implementation is restricted to the case where $g_j(z) = (g(z))^{j-1} \bmod f(z)$, for $j = 1, \dots, s$. In this case, the two polynomials $f(z)$ and $g(z)$ completely describe P_n. This corresponds to *Korobov polynomial rules* [26], and the parameters given in [37] satisfy this restriction.

2.2.8 Polynomial Korobov Rules

This method has been described in the previous subsection and is discussed in more details in [26]. Our current implementation is based on the pseudocode provided in [35] and is restricted to the base 2 case. The user can choose any primitive polynomial over $\mathbb{F}_2$ for $f(z)$, while $g(z)$ is assumed to have the form $g(z) = z^\nu \bmod f(z)$ for some integer $\nu > 1$ relatively prime with $2^d - 1$, where $d = \deg(f(z))$, and such that $\nu \leq k - q$, where q is the degree of the second largest term in $f(z)$ – e.g., if $f(z) = z^7 + z^3 + 1$, then $k - q = 7 - 3 = 4$..

We have also implemented *combined* rules, which are widely used for constructing pseudorandom number generators (see e.g., [16, 22, 46, 45]) with good theoretical and statistical properties. The user is allowed to combine up to three pairs of polynomials $(f_j(z), g_j(z))$ satisfying the conditions outlined previously (and the additional requirement that the $f_j(z)$ have relatively prime degrees). The point set P_n is then obtained by performing a direct sum of the point sets obtained from the individual components based on each pair $(f_j(z), g_j(z))$. Parameters defining "good" polynomial Korobov rules are given in Table 1. The criterion that was used to select these parameters is $\Delta_{32,24,12,8}$, which was mentioned in Section 2.2.2. See [26] for more details and for more parameters.

2.2.9 (Modified) Latin Hypercube Sampling

Our package provides an implementation of the Latin Hypercube Sampling method [28], and a modified version faster to generate but with less theoretical advantages [24]. Although these methods do not really fit the definition of QMC methods, we included them because

Table 1: Parameters for polynomial Korobov rules: each component is described by the polynomial f given in decimal notation, and the integer ν

n	Comp. 1	Comp. 2	Comp. 3
1024	11, 1	137, 4	-
2048	37, 3	67, 5	-
4096	37, 1	131, 2	-
8192	67, 5	131, 4	-
16384	11, 1	19, 1	137, 2
32768	11, 2	37,3	131, 2

they are sometimes tested against Monte Carlo or QMC methods. The original version consists in generating s permutations of the integers from 0 to $n-1$, denoted by $\pi_1, \ldots, \pi_s$. Also required are n randomly and uniformly generated vectors $\Delta_1, \ldots, \Delta_n$ in $[0, 1/n)^s$. The i^{th} point of P_n is defined as

$$\mathbf{u}_i = \left(\frac{\pi_1[i]}{n} + \Delta_{i,1}, \ldots, \frac{\pi_s[i]}{n} + \Delta_{i,s} \right),$$

for $i = 0, \ldots, n-1$. In the modified version, only one random vector Δ uniformly generated over $[0,1)^s$ is required, and for $i = 0, \ldots, n-1$,

$$\mathbf{u}_i = \left(\left(\frac{\pi_1[i]}{n} + \Delta_1 \right) \bmod 1, \ldots, \left(\frac{\pi_s[i]}{n} + \Delta_s \right) \bmod 1 \right).$$

2.2.10 Generic Digital Net

Currently, this method reads the generating matrices of a digital net from a file in an arbitrary base and constructs the corresponding point set. The implementation assumes that the base is prime.

2.3 Using a Gray code

The points of a digital net in base b can be generated efficiently using a Gray code. This idea was first suggested in [1] for Sobol' sequence, and for other constructions in e.g., [12, 37, 45, 3]. The idea is to replace the digits $(i_0, \ldots, i_{d-1})$ associated with the index i used for defining $\mathbf{u}_i$ by the Gray code $(g_0, \ldots, g_{d-1})$ of i, which satisfies the following property: the Gray code for i and $i+1$ only differ in one position; if c is the smallest index such that $i_c \neq b-1$, then g_c is the digit whose value changes, and it becomes $g_c + 1$ in the Gray code for $i+1$ [45, Theorem 6.6].

2.4 Randomizations

We now describe the randomizations available in our package. We plan to implement additional randomization techniques in future versions of the package, such as those proposed by Matoušek for digital nets [27].

2.4.1 Addition of a shift modulo 1

This method consists in generating a random vector Δ uniformly in $[0,1)^s$, and add it to each point of P_n modulo 1. In other words, the randomized point set $\tilde{P}_n$ is given by

$$\tilde{P}_n = \{\mathbf{u}_i + \Delta \bmod 1, i = 1, \ldots, n\}.$$

This randomization is well suited for the Korobov method and more general integration lattices, but can also be used with other quasi-Monte Carlo point sets [30, 49]. In particular, it yields an unbiased estimator Q_n in all cases. The variance of the estimator Q_n based on $\tilde{P}_n$ when P_n is an integration lattice is analyzed in [20]. This method is referred to as `Add Shift` in our implementation.

2.4.2 Addition of a digital shift in base b

When P_n is a digital net in base b, the counterpart of the previous method is to consider the b-ary expansion of the random vector Δ, and to add it to each point of P_n using arithmetic operations over $\mathbb{Z}_b$ (assuming b is prime). More precisely, if $\Delta = (\Delta_1, \ldots, \Delta_s)$ and

$$\Delta_j = \sum_{l=1}^{\infty} d_{j,l} b^{-l},$$

we compute

$$\tilde{u}_{ij} = \sum_{l=1}^{\infty} ((u_{i,j,l} + d_{j,l}) \bmod b)\, b^{-l},$$

and let $\tilde{P}_n = \{\tilde{\mathbf{u}}_i, i = 1, \ldots, n\}$. This randomization was proposed by Raymond Couture for point sets based on linear recurrences modulo 2 [26]. It is also used in an arbitrary base (along with other more time-consuming randomizations) in [27, 12]. It is suited for digital nets in base b and its application preserves the resolution of any projection as well as the value of the quality parameter t. This method is referred to as `Digital Shift` in our implementation.

2.4.3 Random Linear Scrambling

As for the two preceding methods, this randomization technique [27, 12] is designed for digital nets in an arbitrary prime base. It is inspired by Owen's scrambling, and consists in randomly generating s nonsingular lower-triangular $k \times L$ matrices $\mathbf{L}_1, \ldots, \mathbf{L}_s$ over $\mathbb{Z}_b$, where k is the number of digits to scramble, and L is such that $n \leq b^L$. Also required are s randomly and uniformly generated k-dimensional vectors $\mathbf{e}_1, \ldots, \mathbf{e}_s$ over $\mathbb{Z}_b$. The randomization proceeds as follows:

$$\begin{pmatrix} \tilde{u}_{i,j,1} \\ \vdots \\ \tilde{u}_{i,j,L} \end{pmatrix} = \mathbf{L}_j \begin{pmatrix} u_{i,j,1} \\ \vdots \\ u_{i,j,L} \end{pmatrix} + \mathbf{e}_j,$$

where all operations are performed over $\mathbb{Z}_b$. Variants of this randomization are discussed in [27, 12].

3 A Shell for RandQMC programs

We now describe how to use the `RandQMC` package with a program shell. This shell can be used to handle many different integration problems via the general function `function()` defined by the user.

```
/* a RandQMC program shell*/

#include "qmc.h"

#define NUMBLOCKS 2
#define INTLOOP   0
#define EXTLOOP   1

int main (int argc, char **argv)
{
  int i, j;

  double *point = NULL;
  double *rvector = NULL;
  double *rpoint = NULL;
  double result = 0.0;
  int n, m, s;

  STATSQMC stats = NULL;
  CONFIDENCEINTERVAL cf;

  LoadInputQMC(argv[1]);

  InitQMC ();

  stats = InitStat (NUMBLOCKS);

  m = (int) GetQMC(QMC_RANDS);
  n = (int) GetQMC(QMC_NPOINTS);
  s = (int) GetQMC(QMC_DIMEN);
```

```
  for (i = 0; i < m; i++)
  {
    ResetStat (stats, INTLOOP);
    ResetQMC();
    rvector = GenRandom();

    for (j = 0; j < n; j++)
    {
      point = QMC();
      rpoint = AppRandom (point, rvector);
      result = function (rpoint, ...); // a user defined function
      StatUpdate1 (stats, INTLOOP, result);
    }
    StatUpdate1 (stats, EXTLOOP, StatAverage (stats, INTLOOP, RANDVARONE));
  }

  printf("Average = %f", StatAverage(stats, EXTLOOP, RANDVARONE));
  printf("Variance = %f", StatVariance(stats, EXTLOOP, RANDVARONE));
  cf = StatConfIntval (stats, EXTLOOP, RANDVARONE, 0.95);
  printf("Confidence Interval (95%%) = %f to %f", cf.low, cf.high);

  FreeQMC();
  FreeStat (stats);

  return (1);
}
```

The user must provide the information on the method to use and its parameters by creating a file that is passed as an argument when the program is executed. These parameters can then be retrieved by using the function `GetQMC(int index)`, where `index` specifies which parameter is to be retrieved, according to the following convention:

- `0, QMC_METHOD`: The QMC method to use.
- `1, QMC_DIMEN`: The dimension of the point set.
- `2, QMC_NPOINTS`: The number of points in the point set.
- `3, QMC_RMETHOD`: The randomization method to use.
- `4, QMC_RANDS`: The number of randomizations to use.
- `5, QMC_RBASE`: The base to use for the randomization (if applicable).
- `6, QMC_NPARAMS`: The number of extra parameters required to completely specify the method to be used.

In Section 4, we give which integer is associated to each QMC/randomization method (i.e., the value taken by `QMC_METHOD` and `QMC_RMETHOD` in the list above, respectively). However it is usually not necessary to know these numbers since the program `makeinput` automatically generates input files that are consistent with these conventions.

The program shell starts by including the RandQMC library header file, `qmc.h`. All RandQMC programs must include this file in order to work properly. Next, we have three preprocessor defines that are used to keep statistics. The first defines the number of blocks needed (two in our case). The second and third define names for these two blocks.

In the `main()` function we need several variables. First, the two looping variables, `i` and `j`. These are used to loop over the number of randomizations and the number of points in the point set, respectively. Second, we have three pointers of type `double`. They are `point` for the point generated by the QMC method, `rvector` for a random vector to randomize the point, and `rpoint` for the randomized point. The variable `result` is used to store the result of the user defined function. The variables `n, m,` and `s` are integers used to store the number of points, the number of randomizations, and the dimension, respectively. The two remaining variables are `stats` of type `STATSQMC` and `cf` of type `CONFIDENCEINTERVAL`. The first one is where the two blocks `INTLOOP` and `EXTLOOP` are stored. The second one is the confidence interval returned by one of the `stats` functions.

The first instruction in the program is to call the function `LoadInputQMC()`, which reads the input file and gets the information required to generate the randomized QMC point set. Next we must initialize the QMC and randomization methods, and the statistics accumulator. This is accomplished by the function calls to `InitQMC()`, and `InitStat()`.

The exterior `for` loop runs over the number of randomizations. The next function calls we have are `ResetQMC()` to reset the point set, and `ResetStat()` to reset the statistical data block `INTLOOP` used in the interior `for` loop. Next, the function `GenRandom()` generates a random vector to randomize the point set. It returns a pointer to a random vector, and we save it in `rvector`.

The interior `for` loop will run over the entire point set generating one point at a time. The function `QMC()` generates this point, and returns a pointer to it. This pointer is saved in the variable `point`. We randomize the point with a call to `AppRandom()`. It takes the pointer, `point`, and randomizes it with the random vector previously generated (saved in `rvector`). `AppRandom()` returns a pointer to the randomized point, which is saved in `rpoint`. We now call the user defined function with the randomized point. The result of the function is used to update the interior `for` loop statistical data block, `INTLOOP`. We update this block over the entire point set, and then update the exterior `for` loop statistical data block, `EXTLOOP`, with the average.

When all the randomizations are done the average, variance, and a confidence interval for the `EXTLOOP` block are printed. The integer `RANDVARONE` is a predefined constant that specifies for which one of possibly two random variables the user wants to retrieve the average, variance, and confidence interval. The last function calls are to free the QMC and randomization methods, and the statistics accumulator. This is done by the last two functions `FreeQMC()`, and `FreeStat()`.

This shell is used in the example programs given in Appendix A, and can serve as a basis for many other applications, the user's task simply being to write the appropriate `function()` procedure. The next two sections describe the input file and a program that can be used to generate it.

4 The Input File

This section describes the format of the input file for the `qmc` library. This file contains all the input needed by the methods, and it has seven major entries in it. Although a program (`makeinput`) to make the input file is part of this package, this section describes precisely what each entry in the file means so that the user can customize the input beyond the ability of the `makeinput` program.

The file contains seven input parameters to the program. Each input is a double and each parameter is separated by a newline character. A description of each of the seven values in the file is given below.

1. `Method`: The method is the first entry in the input file. Each method is defined by a number as follows.
   - `Monte Carlo`: 1
   - `Korobov Method`: 2
   - `Polynomial Korobov Method`: 3
   - `Sobol' Method`: 4
   - `Shift Net Method`: 5
   - `Generalized Faure Method (Tezuka and Tokuyama 94)`: 6
   - `Generalized Faure Method (Faure 01)`: 7
   - `Faure Method`: 8
   - `Halton Sequence`: 9
   - `Generalized Halton Sequence`: 10
   - `Salzburg Tables`: 11
   - `Latin Hypercube Sampling`: 12
   - `Modified Latin Hypercube Sampling`: 13
   - `Generic Digital Net`: 14
2. `Dimension`: The dimension is the second entry in the input file. The admissible range for each method can be found in Section 6.
3. `The number of points`: The next entry in the input file is the number of points. The correct values for each method can be found in Section 6.
4. `Randomization method`: Like the point generation methods, each randomization method is defined by a number.
   - `No Randomization`: 1
   - `Add Shift`: 2

- `Digital Shift`: 3
- `Random Linear Scrambling`: 4

5. `Number of randomizations`: This is the number of times that the point set will be randomized and passed to the function point by point. This can be any number between 0 and $2^{31} - 1$.

6. `Randomization base`: This is the base over which the randomization is defined. It is used by the digital shift and the two scrambling methods. For `Add Shift` the base is irrelevant, and can therefore be set to 2.

7. `Number of extra parameters`: This is the number of extra parameters (if any) required to describe the chosen method. Each method may or may not use them; we refer the user to Section 6 to see how they are used for each method.

5 The `makeinput` Program

The `makeinput` program is used to generate input files for QMC programs. It can make an input file for all the methods that are supported in this library. The `makeinput` program will prompt the user for all of the data that it needs. There are limitations to the type of input that can be generated by the program; to customize the input file for a given method see Section 4 and the method's description in Section 6. An example illustrating how to run the `makeinput` program is given below.

```
[qmc]$ makeinput samplefile

        QMC Input File Generator

QMC methods to use:
(1)  MONTE CARLO
(2)  KOROBOV
(3)  POLYNOMIAL KOROBOV
(4)  SOBOL
(5)  SHIFTNET
(6)  GENERALIZED FAURE, TEZUKA and TOKUYAMA 94
(7)  GENERALIZED FAURE, FAURE 01
(8)  FAURE
(9)  HALTON SEQUENCE
(10) GENERALIZED HALTON SEQUENCE
(11) SALZBURG TABLES
(12) LATIN HYPERCUBE SAMPLING
(13) MODIFIED LATIN HYPERCUBE SAMPLING
(14) GENERIC DIGITAL NET
Choose a method between 1 and 14: 4
```

```
Enter dimension: 30

Enter the number of points to generate: 32768

Randomization method to use:
(1) NO SHIFT
(2) ADD SHIFT
(3) DIGITAL SHIFT
(4) RANDOM LINEAR SCRAMBLING
Choose(1-4): 3

Enter number of randomizations: 10
[qmc]$
```

After running the above sample the output is sent to a file called `samplefile`. The contents of this file are shown below.

```
4
30
32768
3
10
2
0
```

For a detailed description of each of the entries see Section 4.

6 The Methods

The program shell in Section 3 uses four functions to generate a QMC point set, which we now describe. Three of these functions are function pointers. They are pointed to an actual function by `InitQMC()`, which is not a function pointer. In addition to these, the function `LoadInputQMC()` is used to read the input file, and `GetQMC()` can be used to retrieve information on the point set being constructed and the randomization method that has been chosen.

`int LoadInputQMC (char * fname);`

This function reads the file `fname` and stores its information so that `InitQMC` will work properly. It therefore has to be called before `InitQMC`.

`int InitQMC (void);`

This function initializes the correct point set generator as specified by the input file read by `LoadInputQMC`.

`double *QMC (void);`

This function returns one point from the point set.

`void ResetQMC (void);`

This function resets the generator, so that the point set can be randomized again.

`void FreeQMC (void);`

This function must be called at the end of a program, or when the point set generator is no longer needed. It does the same thing for each method.

`double GetQMC(int index);`

This function returns a parameter (specified by `index`) describing the point set or the randomization. See Section 3 for more details.

Next, we provide input parameters for each method along with a sample input file. The order in which the parameters appear here is the same as their order in the input file. We also include the descriptions of the actual functions mentioned earlier, and some sample output.

6.1 Monte Carlo Method

6.1.1 Input Parameters

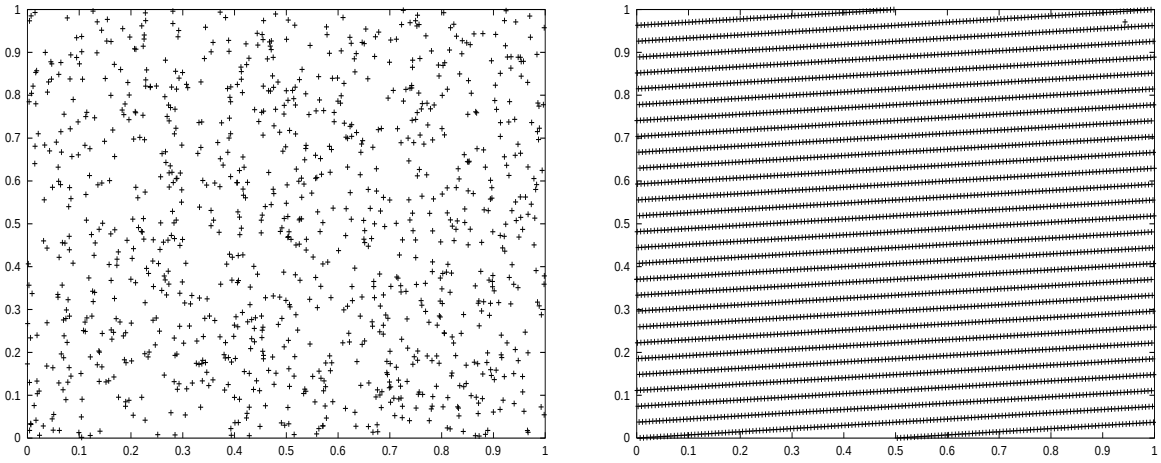
- `Method`: This is defined as the value 1.
- `Dimension`: This can be any value between 1 and $2^{31} - 1$.
- `Number of points`: This is the number of points that will be generated. It can be a number between 0 and $2^{31} - 1$.
- `Randomization method`: The Monte Carlo method uses independent, uniform, and random points, so there is no randomization required.
- `Number of randomizations`: Specifying this number to be m means that m i. i. d. copies of an estimator based on n points will be generated, where n is the number of points in the point set.
- `Randomization base`: This is irrelevant and therefore defined as 2.
- `Number of extra parameters`: There are no extra parameters.

An example of an input file that will generate 10000 points in 30 dimensions is given in Figure 1.

Figure 1: Input file example for Monte Carlo

```
1
30
10000
2
1
2
0
```

Figure 2: Left: 2-dim. Monte Carlo point set with 1024 points; Right: 2-dim. Korobov point set with 2039 points.



6.1.2 Functions

`int InitMC (int s);`

This function allocates the memory for one point in the point set. It returns 1 on success or 0 on failure. The only reason for failure is that there is not enough memory.

`double *MC ();`

This function generates a random point. It is returned as a pointer of type `double`. If `NULL` is returned then `InitMC()` has not been called.

`void ResetMC ();`

This procedure does not do anything since the method cannot be randomized.

`void FreeMC ();`

This procedure returns the memory allocated by `InitMC()` to the operating system.

6.2 Korobov Method

6.2.1 Input Parameters

- `Method`: This is defined as the value 2.
- `Dimension`: This can be any number between 1 and $2^{31} - 1$.
- `Number of points`: The number of points allowed can be either a prime number or a power of 2 between 1 and $2^{31} - 1$.
- `Randomization method`: The most appropriate method is `Add Shift`.
- `Number of randomizations`: This can be a value between 0 and $2^{31} - 1$.
- `Randomization base`: For the Korobov method this is irrelevant and therefore defined as 2.
- `Number of extra parameters`: There are two extra parameters.
- `Generator` a: This is the first of the two extra parameters required by this method. If the number of points is a power of 2 then a must be an odd number.
- `Type`: This is one of the following:
  - If the number of points, n, is prime and a is a primitive element modulo n then `Type` $= 1$.
  - If n is prime and a is not a primitive element modulo n then `Type` $= 2$.
  - If n is a power of two and a is odd then `Type` $= 2$.

An example of an input file that generates 4093 points in dimension 30 using the `Add Shift` with 50 randomizations is given in Figure 3. Since the number of points is prime (4093) and the generator $a = 1516$ is a primitive element, `Type` is defined as 1.

6.2.2 Functions

`int InitKorobov (int s, int n, int a, int type);`

This function initializes the Korobov generator, and allocates the memory required for a point in the dimension specified by the parameter `s`. The parameter `n` is the number of points requested, `a` is the generator and `type` is the category of the generator. The function returns 1 on success and 0 on failure. If 0 is returned then there is not enough memory for the generator.

`double *Korobov ();`

This function returns a point from the point set. That is, it returns a pointer to an array of type `double` that contains the point. If an error occurs the function returns a `NULL` pointer.

Figure 3: Input file example for Korobov

```
2
30
4093
2
50
2
2
1516
1
```

The first possibility for error is that `InitKorobov()` was not called. The second possibility is that the number of points specified has been exceeded. The final possibility for error is that the last parameter of `InitKorobov()` is neither 1 nor 2. Refer to the Input Parameters section to fix this. The first point returned is always (0,0,0,...,0_s).

```
void ResetKorobov ();
```

This function resets the generator, so that `Korobov()` can once again generate `n` number of points.

```
void FreeKorobov ();
```

This function frees all the memory allocated by `InitKorobov()`.

6.3 Polynomial Korobov

6.3.1 Input Parameters

- `Method`: This is defined as the value 3.
- `Dimension`: This can be any number between 1 and $2^{31} - 1$.
- `Number of points`: The number of points should be a power of 2 between 1 and 2^{30}.
- `Randomization method`: This can be any of the defined methods, but a digital shift is more appropriate. The random linear scrambling has not been implemented.
- `Number of randomizations`: This can be a value between 0 and $2^{31} - 1$.
- `Randomization base`: For the polynomial Korobov method this is defined as 2 because our implementation assumes the method is defined over base 2.
- `Number of extra parameters`: There are at least three extra parameters, with a maximum of seven.

- `Number of components`: This is the first of the extra parameters required by this method. It represents the number of simple generators that will be combined. Its value can be 1, 2 or 3, and determines the number of extra parameters because each component requires two parameters, which we now describe:
- `Characteristic polynomial`: This is the characteristic polynomial $f(z)$ of the recurrence of a simple generator. It must be a primitive polynomial over $\mathbb{F}_2$, and is given in decimal notation by the user.
- `Step parameter`: This corresponds to the parameter ν in Section 2.2.8. Please note that `makeinput` **will not** verify that ν satisfies the conditions outlined in that section, so it is up to the user to make sure that they do. The parameters given in Table 1 satisfy these constraints.

An example of an input file that generates 2048 points in dimension 32 based on two components given by $f(z) = z^4 + z + 1$ and $\nu = 1$, and $f(z) = z^7 + z^3 + 1$ and $\nu = 4$, and using 10 digital shifts is given in Figure 4. Since there are two components, the number of extra parameters (seventh entry) is 5.

Figure 4: Input file example for polynomial Korobov

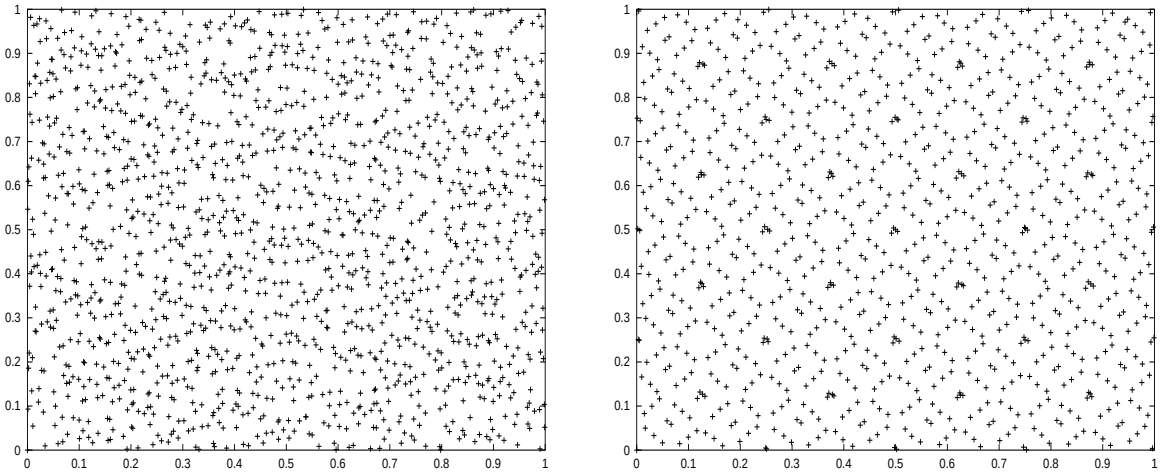
```
3
32
2048
3
10
2
5
2
19
1
137
4
```

6.3.2 Functions

```
int InitCpKorobov (int s, double points, int num_gens, double * gen_params);
```

This function initializes the polynomial Korobov method, and allocates the memory required for a point in the dimension specified by the parameter `s`. The parameter `points` is the number of points requested, `num_gens` is the number of components and `genparams` contains the information on these components. The function returns 1 on success and 0 on failure. If 0 is returned then there is not enough memory for the generator.

Figure 5: Left: 2-dim. 2-dim. polynomial Korobov point set with 2048 points; Right: 2-dim. Sobol' point set with 2039 points.



`double *CpKorobov ();`

This function returns a point from the point set. That is, it returns a pointer to an array of type `double` that contains the point. If an error occurs the function returns a `NULL` pointer. The first possibility for error is that `InitCpKorobov()` was not called. The second possibility is that the number of points specified has been exceeded. The first point returned is always $(0,0,0,...,0_s)$.

`void ResetCpKorobov ();`

This function resets the generator, so that `CpKorobov()` can once again generate the specified number of points.

`void FreeCpKorobov ();`

This function frees all the memory allocated by `InitCpKorobov()`.

6.4 Sobol' Method

6.4.1 Input Parameters

- `Method`: This is defined as the value 4.
- `Dimension`: This can be any number between 1 and 360.
- `Number of points`: This can be any number less that $2^{31} - 1$, but the Sobol' method works best when the size of the point set is a power of 2 or a multiple of a power of 2.

- `Randomization method`: This can be any of the defined methods, but a digital shift or scrambling is more appropriate.
- `Number of randomizations`: This can be between 0 and $2^{31} - 1$.
- `Randomization base`: For the Sobol' method this is always defined as 2.
- `Number of extra parameters`: There are no extra parameters.

Figure 6 gives an example file that will use the Sobol' method to generate 32768 points in dimension 20 with no randomization.

Figure 6: Input file example for Sobol'

```
4
20
32768
1
1
2
0
```

6.4.2 Functions

`int InitSobol (int s, unsigned int max);`

This function initializes the Sobol' generator that generates a point set of dimension `s`. The parameter `max` is the maximum number of points that can be generated. It returns 1 on success and 0 on failure. If 0 is returned then one possibility of failure is that the dimension of the point set is not between 1 and 360. Check that the parameter `s` satisfies $1 \leq s \leq 360$. Another possibility of failure is that `n` is greater than $2^{31} - 1$. The final possibility for failure is if there is not enough memory.

`double *Sobol ();`

This function returns one point from the point set. If the function returns `NULL` then either `InitSobol()` has not been called or the function has been called more times than the number of points. The first point returned is always $(0,0,0,...,0_s)$.

`void ResetSobol ();`

This function resets the Sobol' generator, so that it will once again be able to generate `n` number of points.

`void FreeSobol ();`

This function frees all the memory that was allocated by the `InitSobol()` function.

6.4.3 Changing Sobol' Direction Numbers

In our implementation, the direction numbers are stored as a two-dimensional `unsigned integer` array of size 360 by 12 (since the degree of the 360^{th} primitive polynomial over $\mathbb{F}_2$ is 12) called `minit`. The j^{th} row of the array corresponds to a matrix for the j^{th} dimension, and it contains $m_{j,1} \ldots m_{j,12}$ (where $m_{j,l} = 0$ if l is larger than the degree of the l^{th} primitive polynomial). In order to change the current values of the array, open the file `soboldata.h` and modify `minit[][]`. Note that the library must be recompiled if the file `soboldata.h` changes, and therefore the source code is required to do that.

6.4.4 Quality Parameter t

The following table, taken from [32], gives the values of the quality parameter t such that Sobol' sequence is a (t, s)-sequence, for $1 \leq s \leq 20$.

Table 2: Values of t for Sobol' sequence

s	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
t	0	0	1	3	5	8	11	15	19	23	27	31	35	40	45	50	55	60	65	71

6.5 Shift-net Method

6.5.1 Input Parameters

- `Method`: This is defined as the value 5.
- `Dimension`: This can be a number between 3 and 32.
- `Number of points`: The number of points should be 2 raised to a power smaller or equal to the dimension. In theory, one can use propagation rules to define nets of larger sizes, but implementing this in a trivial way can introduce bad properties (e.g., superposition of points on lower-dimensional projections). Hence the `makeinput` program prevents the user from making such a choice.
- `Randomization method`: This can be any method allowed, but not the random linear scrambling (has not been implemented yet).
- `Number of randomizations`: This can be any number between 0 and $2^{31} - 1$.
- `Randomization base`: The base is 2 in our implementation.
- `Number of extra parameters`: There are no extra parameters.

An example of an input file that generates 1024 points in dimension 10 with 23 scramblings is given in Figure 7.

Figure 7: Input file example for shift-net

```
5
10
1024
4
23
2
0
```

6.5.2 Functions

`int InitShiftNet (int s, unsigned int n);`

This function allocates the memory for the point in the requested dimension. The parameter `n` is the number of points and the parameter `s` is the dimension. This function returns 1 upon success and 0 upon failure. The first possibility of failure is that `s` does not satisfy $3 \leq$ `s` ≤ 32. The second possibility is that the number of points `n` is larger than 2 raised to the dimension. The final possibility of error is that there is not enough memory.

`double *ShiftNet ();`

This function returns one point from the point set. The point is returned as a pointer of type `double`. If the number of calls to this function exceeds the number of points then a `NULL` pointer is returned. The first point returned is always (0,0,0,...,0_s).

`void ResetShiftNet ();`

This function resets the parameters in the Shift Net method, so that the function `ShiftNet()` can once again be called `n` number of times.

`void FreeShiftNet ();`

This function is to be called when the program is done or when `ShiftNet()` is no longer needed. It frees all of the memory allocated by `InitShiftNet()`.

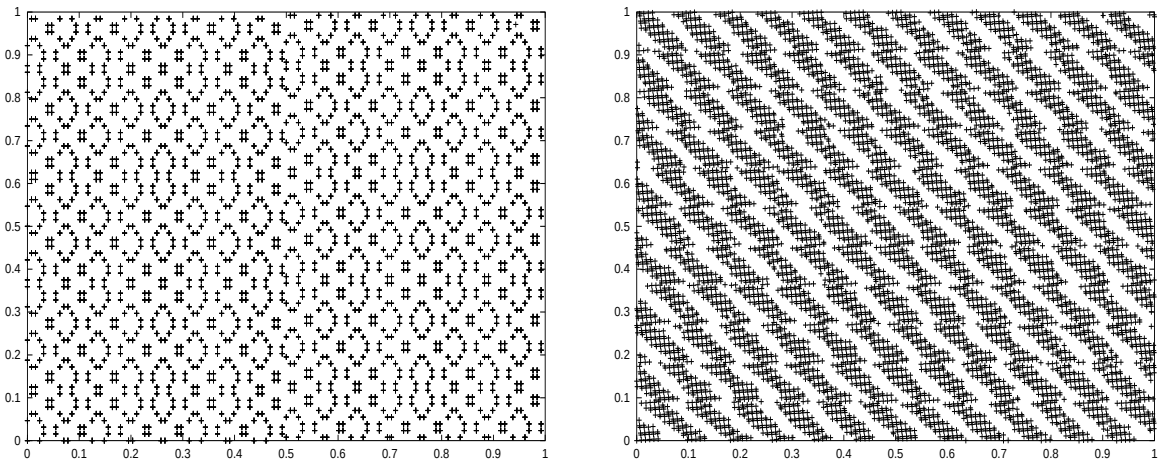
6.5.3 Quality Parameter t

The shift-nets implemented in this package are based on the parameters given in [38]. In Table 3 below, we reproduce the values of the quality parameter t as given in that paper.

Table 3: Quality parameter for shift-nets

s	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	$23 \leq d \leq 39$
t	0	1	1	2	2	3	3	4	4	5	6	6	7	8	8	9	10	10	11	12	$d-10$

Figure 8: Left: 2-dim. point set from a shift-net with 8192 points; Right: 2-dim. generalized Faure sequence of T. & T. with 8192 points.



6.6 The Faure Methods

Three Faure methods have been implemented. They are the Generalized Faure method by Tezuka and Tokuyama [47], the Generalized Faure method by Faure [8], and the original Faure method [7]. All of these methods use the same input parameters, so they have been put together into one section.

6.6.1 Input Parameters

- `Method`: This is defined as the value 6 for the generalized Faure method by Tezuka and Tokuyama, the value 7 for the generalized Faure method by Faure, and the value 8 for the Faure method.

- `Dimension`: This can be any number between 1 and $2^{31} - 1$. However, it has only been tested for dimension smaller or equal to 360. Also, the `makeinput` program has an upper bound of 360 on the dimension and therefore, if the dimension is larger than 360 this program should be modified, or the input file should be created manually.

- `Number of points`: This can be any number between 0 and $2^{31} - 1$. However, the Faure methods work best when the number of points is a power of the base (described below), or a multiple of a power of the base.

- `Randomization method`: This can be any method allowed, but scrambling or digital shift are better suited.

- `Number of randomizations`: This can be any number between 0 and $2^{31} - 1$.

- `Randomization Base`: This is the same value as the base for the generator.
- `Number of extra parameters`: There is one extra parameter.
- `Base`: The base is the only extra parameter. It is set to the smallest prime number greater than or equal to the dimension. The `makeinput` program can find this value for a dimension smaller or equal to 360.

An example of an input file that generates 29791 points using the Faure method in dimension 30 with base 31 is given in Figure 9. The randomization is achieved by 20 digital shifts in base $b = 31$.

Figure 9: Input file example for Faure

```
8
30
29791
3
20
31
1
31
```

6.6.2 Functions

`int InitFaure (int s, int base, int method, int n);`

This function allocates the memory for the point in the requested dimension. The parameters are described as follows: `s` is the dimension, `base` is the base, `method` is 6, 7, or 8 depending on the method chosen, and `n` is the number of points. Upon success the function returns 1, else it returns 0. One possibility for a return value of 0 is that the dimension is less than 1 or greater than a reasonable amount to handle. A second possibility is that `base` is less than `s` (notice `base` is not checked for being prime). A third possibility is that `method` does not satisfy, $6 \leq$ `method` ≤ 8. The final possibility is that there is not enough memory.

`double *Faure ();`

This function returns a point from the point set. It returns a pointer to an array of `double` that contains the point. If `NULL` is returned the generator was unable to produce a point. This only happens if the number of calls to `Faure()` exceeds the number of points. The first point returned is always $(0,0,0,...,0_s)$.

`void ResetFaure ();`

This function resets the parameters in all of the methods. After it is called `Faure()` can once again produce `n` number of points. This procedure is faster than calling `FreeFaure()` and then `InitFaure()`.

`void FreeFaure ();`

This is to be called when the program is done or when the generator is no longer needed. It frees all the memory allocated by `InitFaure()`.

6.6.3 Changing the Faure Matrices

This section describes how to change the generating matrices for the three Faure methods. There is a function called `GenMatrices()` in the file `genmat.c` (available from the authors) where the matrices are generated, and stored in the array `matrices`. Hence, to change the matrices this function must be modified. In the function, the variable `pascalmatrix` is the Pascal matrix that is raised to the power of each dimension less than or equal to the requested dimension. Each resulting matrix is temporarily stored in the variable `cmatrix`, so that it can be manipulated. Only after the function is done with `cmatrix` will it copy the entries to the function parameter `matrices`. This parameter contains the generating matrices the Faure methods will use.

6.7 Halton Sequence

6.7.1 Input Parameters

- `Method`: This is defined as the value 9.
- `Dimension`: This can be any number between 1 and 360.
- `Number of points`: The number of points the Halton Sequence can generate is restricted to $2^{32} - 1$.
- `Randomization method`: We recommend the `Add Shift`.
- `Number of randomizations`: This can be any number between 0 and $2^{31} - 1$.
- `Randomization base`: The Halton Sequence uses a different base for each dimension making this value irrelevant, and therefore it is defaulted to 2.
- `Number of extra parameters`: There are no extra parameters.

An example of an input file that generates 16384 points using the Halton Sequence method in dimension 15 with the `Add Shift` and 20 randomizations is given in Figure 10.

Figure 10: Input file example for Halton

```
9
15
16384
2
20
2
0
```

6.7.2 Functions

`int InitHalton (int s);`

This function allocates the memory for the point in the requested dimension. The parameter `s` is the dimension to use. Upon success the function returns 1, else it returns 0. If 0 is returned then the first possible error is that `s` does not satisfy $1 \leq s \leq 360$. The only other error is if there is not enough memory for the generator.

`double *Halton ();`

This function returns a point from the point set. If more than $2^{32} - 1$ points are generated then the point is incorrect, and there is an overflow error. The first point returned is always $(0,0,0,...,0_s)$.

`void ResetHalton ();`

This function resets the Halton Sequence generator to the first point in the sequence. After it is called, `Halton()` can once again produce `n` number of points. This procedure is faster than calling `FreeHalton()` and then `InitHalton` because memory does not have to be re-allocated.

`void FreeHalton ();`

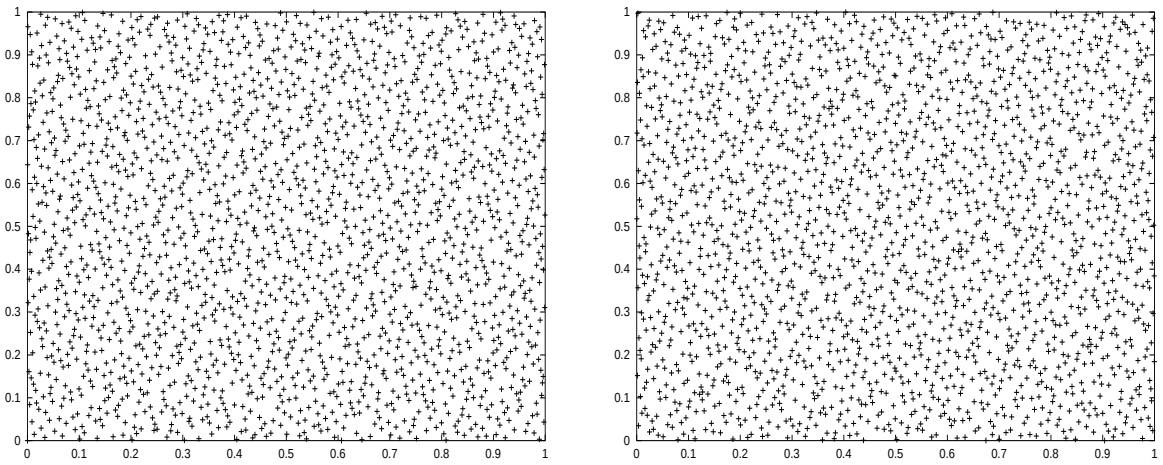
This function is to be called when the program is done or when the generator is no longer needed. It frees all the memory allocated by `InitHalton()`.

6.8 Generalized Halton Sequence

6.8.1 Input Parameters

- `Method`: This is defined as the value 10.
- `Dimension`: This can be any number between 1 and 360.
- `Number of points`: The number of points the Halton Sequence can generate is restricted to $2^{32} - 1$.

Figure 11: Left: 2-dim. Halton Sequence with 2048 points; Right: 2-dim. point set from the generalized Halton sequence with 2048 points.



- `Randomization method`: This method should not be randomized further since it is already randomized by definition. Hence the `makeinput` program does not even ask for a randomization method in this case.

- `Number of randomizations`: This can be any number between 0 and $2^{31} - 1$.

- `Randomization base`: The Halton Sequence uses a different base for each dimension making this value irrelevant, and therefore it is defaulted to 2.

- `Number of extra parameters`: There are no extra parameters.

An example of an input file that generates 16384 points using the Generalized Halton Sequence method in dimension 15 and with 20 randomizations is given in Figure 12.

Figure 12: Input file for generalized Halton

```
10
15
16384
1
20
2
0
```

6.8.2 Functions

`int InitGHalton (int s);`

This function allocates the memory for the point in the requested dimension. The parameter `s` is the dimension to use. Upon success the function returns 1, else it returns 0. If 0 is returned then the first possible error is that `s` does not satisfy $1 \leq s \leq 360$. The only other error is if there is not enough memory for the generator.

`double *GHalton ();`

This function returns a point from the point set. If more than $2^{32} - 1$ points are generated then the point is incorrect, and there is an overflow error. The first point returned is always $(0,0,0,...,0_s)$.

`void ResetGHalton ();`

This function resets the Halton Sequence generator to the first point in the sequence. After it is called, `Halton()` can once again produce `n` number of points. This procedure is faster than calling `FreeHalton()` and then `InitHalton` because memory does not have to be re-allocated.

`void FreeGHalton ();`

This function is to be called when the program is done or when the generator is no longer needed. It frees all the memory allocated by `InitGHalton()`.

6.9 Salzburg Tables

6.9.1 Input Parameters

- `Method`: This is defined as the value 11.
- `Dimension`: When running the `makeinput` program, the user has the choice between specifying its own polynomials (using a prime base), or choosing from the tables in [37]. If the latter option is chosen, then the dimensions available for each (base, degree) pair are given in Tables 4 and 5, which come from [37]. For a given number of points, these tables give the dimensions for which an optimal polynomial has been given in [37], and its associated value for the quality parameter t in parentheses.
- `Number of points`: The number of points this method can generate is restricted to $2^{32} - 1$. However, the number of points should be $base^{degree}$.
- `Randomization method`: This can be any method, but a digital shift or scrambling is more appropriate.
- `Number of randomizations`: This can be any number between 1 and $2^{31} - 1$.
- `Randomization base`: This is the same as the base over which the point set is defined.

- `Number of extra parameters`: There are four extra parameters.
- `Degree of` f: This is the first of the four extra parameters required for this method. It is the degree of the polynomial f and corresponds to the number of rows of the generating matrices.
- `Base`: This is the base over which the point set is defined. It must be a prime number and is restricted to 3 or 5 in the case where the user wants to choose parameters from the tables in[37].
- `First polynomial`: This is the polynomial $f(z)$, given in decimal notation as in [37].
- `Second polynomial`: This is the polynomial $g(z)$, given in decimal notation as in [37].

Table 4: Base 3

Degree	Dimension				
14	28(8)	29(8)	30(8)	31(8)	32(8)
15	28(9)	29(9)	30(9)	31(9)	32(9)
16	28(9)	29(9)	30(9)	31(9)	32(10)
17	28(10)	29(10)	30(10)	31(10)	32(10)
18	28(10)	29(11)	30(11)	31(11)	32(11)

Table 5: Base 5

Degree	Dimension			
14	28(7)	29(7)	30(7)	31(7)
15	28(7)	29(8)	30(8)	31(8)
16	28(8)	29(8)	30(8)	31(8)
17	28(9)	29(9)	30(9)	31(9)

An example of an input file that generates 19683 points using polynomials from the Salzburg Tables[37] in dimension 30 is given in Figure 13. In this case the base is 3, the degree of f 14, and random linear scrambling is applied 10 times.

6.9.2 Functions

```
int InitSalzburg (int s, int b, int m, double poly_f, double poly_g);
```

This function allocates the memory for the point in the requested dimension. The parameter `s` is the dimension to use, `b` is the base to use, `m` is the degree of `poly_f`, the first polynomial, and `poly_g` is the second polynomial ($g(z)$ in Section 2). Upon successful initialization the function returns 1, otherwise it returns 0. If 0 is returned then the only possible error is that there is not enough memory for the generator.

Figure 13: Input file for Salzburg Table

```
11
30
19683
5
10
3
4
3
17
25579194
667496
```

`double *Salzburg ();`

This function returns a point from the point set. This is a pointer to an array of type `double`. If `NULL` is returned then an error has occured. The only reason this can happen is if the number of calls to this function exceeds the number of points. The first point returned is always $(0,0,0,...,0_s)$.

`void ResetSalzburg ();`

This function resets the Salzburg Tables generator so that it can generate `n` number of points again. It is much faster than calling `InitSalzburg()` and then `FreeSalzburg()` to reset the generator.

`void FreeSalzburg ();`

This function is to be called when the program is done or when the generator is no longer needed. It frees all the memory allocated by `InitSalzburg()`.

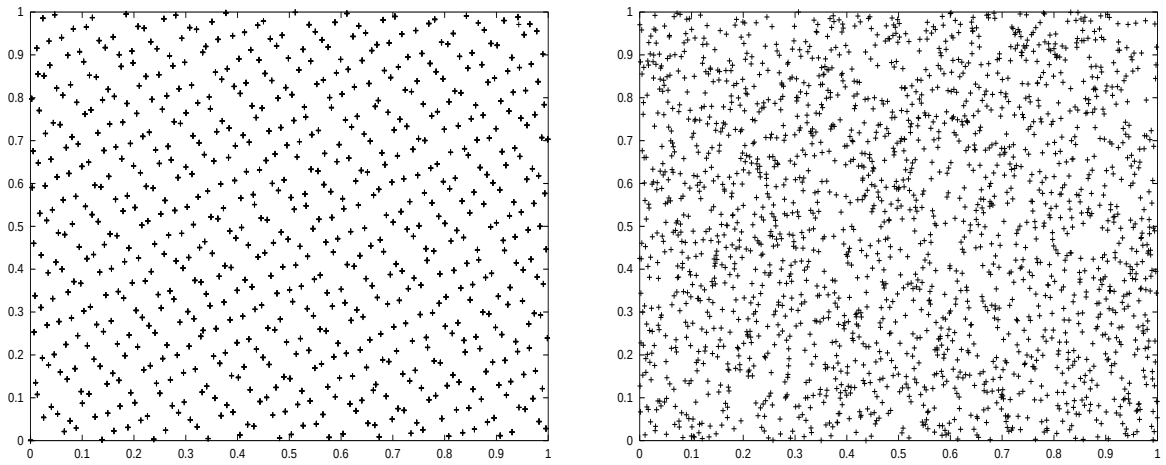
6.10 Latin Hypercube Sampling Methods

There are two Latin Hypercube Sampling methods. The first one is the original LHS method from [28], and the second one is the modified LHS method discussed in Section 2. The difference is irrelevant to the input parameters, except for the randomization method. For this reason the two methods are presented together.

6.10.1 Input Parameters

- `Method`: This is defined as the value 12 and the integer 13 for the LHS method and Modified LHS method, respectively.

Figure 14: Left: 2-dim. point set from the Salzburg Tables with 4096 points in base 3; Right: 2-dim. point set from LHS with 2048 points.



- `Dimension`: This can be any number between 1 and $2^{31} - 1$.
- `Number of points`: The largest number of points these methods can generate is restricted to $2^{31} - 1$.
- `Randomization method`: The LHS method uses its own randomization method, so there is no need to randomize it. Hence, for the LHS method the value should be set to `No Shift`. The Modified LHS method does require randomization, and the random method should be set to `Add Shift`.
- `Number of randomizations`: This can be any number between 1 and $2^{31} - 1$.
- `Randomization base`: The base for these methods is irrelevant, and therefore is set to 2.
- `Number of extra parameters`: There are no extra parameters.

An example of an input file that generates 16384 points using the LHS method in dimension 30 with 10 randomizations is given in Figure 15. Notice that no randomization has been chosen, as required.

6.10.2 Functions

`int InitLHS (int s, int n);`

This function allocates the memory for the point in the dimension specified by the parameter `s`. The other parameter, `n`, is the number of points in the point set. Upon successful initialization the function returns 1, else it returns 0. The only reason a 0 would be returned

Figure 15: Input file for LHS

```
10
30
16384
1
10
2
0
```

is if there was not enough memory for the generator. A point set of 16384 points in dimension 30 requires 1.875 megabytes while a point set of 2^{20} points in dimension 30 requires 120 megabytes. Hence, this error may occur very easily.

```
double *LHS ();
double *MLHS ();
```

These functions return a point from the point set for their respective methods (`LHS()` for the LHS method and `MLHS()` for the Modified LHS method). The point is an array of type `double`, so more precisely a pointer is returned to this array. If `NULL` is returned then an error has occured. The reason for this is that the number of calls to the function exceeded the number of points. The first point returned is (0,0,0,...,0_s) for the Modified LHS method, but is not (0,0,0,...,0_s) for the LHS method.

```
void ResetLHS ();
```

This function resets the method, so that `LHS()` or `MLHS()` can once again generate `n` number of points.

```
void FreeLHS ();
```

This is to be called when the program is done or when the generator is no longer needed. It frees all the memory allocated by `InitLHS()`.

6.11 Generic Digital Net

It is important to notify the user that to use this method, generating matrices for the net must be stored in a file called "matrices.dat".

6.11.1 Parameters

- `Method`: This is defined as the value 14.
- `Dimension`: This can be any number between 1 and $2^{31} - 1$.

- `Number of points`: This can be any number between 0 and $2^{31} - 1$. However, it is preferable to choose a power of the base (described below), or a multiple of a power of the base.
- `Randomization method`: This can be any method allowed, but scrambling or digital shift are better suited.
- `Number of Randomizations`: This can be any number between 0 and $2^{31} - 1$.
- `Randomization base`: This is the same value as the base for the generator.
- `Number of extra parameters`: There are two extra parameters.
- `Number of rows`: This is the number of rows of each generating matrix.
- `Base`: This is the base over which the point set is defined. It should be a prime number and the generating matrices contained in `matrices.dat` should contain numbers in the corresponding ring.

An example of an input file that generates 1024 points using a generic digital net in dimension 64 is given in Figure 16. In this case the base is 2, the number of rows is 10, and a digital shift is applied 10 times.

Figure 16: Input file for Salzburg Table

```
14
64
1024
3
10
2
2
2
10
```

6.11.2 Functions

We only give the initialization function, as the other functions use the Salzburg tables implementation.

`int InitGenericNet (int s, int b, int m);`

This function allocates the memory for the point in the requested dimension. The parameter `s` is the dimension to use, `b` is the base to use, and `m` is the number of rows. Upon successful initialization the function returns 1, otherwise it returns 0. If 0 is returned then the only possible error is that there is not enough memory for the generator.

6.12 Adding QMC Methods

We now explain how to add new QMC methods to the `RandQMC` library. Assume that `method` is the name of the new method.

1. `Create two files: method.h and method.c`
2. `In method.h`: add four function prototypes:
   - `int InitMethod (int s, ...)`: This is the initialization function for the new method. It must allocate memory for one point of the point set. The parameters depend on the method. The parameter `s` is the dimension of the point set. The function has to return 1 if it is successful, and 0 if it is not.
   - `double *Method (void)`: This function should return a point from the point set, and if it cannot then `NULL` should be returned.
   - `void ResetMethod (void)`: This function has to reset the new method so that it can generate the same point set as it did when it was first initialized.
   - `void FreeMethod (void)`: This function has to free any memory the method allocated.
3. `In method.c`:
   - `double *point`: Declare the `static` variable `point`. This is the vector that contains one point from the point set. In `InitMethod()`, memory has to be allocated for it. This can be done by adding the line, (`s` is the dimension)
     ```
     point = (double *) malloc (sizeof(double) * s);
     ```
     The `Method()` function should return the point it generates. This is done by returning the `point` variable, using `return (point)`. In the `FreeMethod()` function the memory that was allocated for the point must be freed. This is done by calling `free (point)`;
   - `int count`: Declare the `static` variable `count`. This variable tracks which point from the point set is being generated. In both `InitMethod()` and `ResetMethod()`, set `count` to 0. Then, to return the point (0,0,0,...,0_s) the first time `Method()` is called, in `InitMethod()` add the following line after the line where memory for `point` is allocated,
     ```
     memset (point, 0, sizeof(double) * s);
     ```
     This will clear the vector to all zeros. Now, the `Method()` function must return the point (0,0,0,...,0_s) for the first time it is called. Since `point` was cleared to all zeros, we can return it as the first point. Therefore, add the following line to `Method()`,
     ```
     if (count == 0)
       return (point);
     ```
     Finally, we must increment count in the `Method()` function. This should be done right before `Method()` returns to its calling function.

4. In `qmc.h`: Add a preprocessor define for the new method. Assuming the new method is associated to the integer 100, it would be something like `#define NEW_METHOD 100`.

5. In `qmc.c`:

   - `#include "method.h"`: Add the preprocessor directive to include the header file for the new method.
   - `int InitQMC ()`: The new method must be added to the `switch` statement of this function. Add the following lines, before the `default` case,

     ```
     case NEW_METHOD:
       retvalue = InitMethod (args[QMC_DIMEN],...);
       QMC = Method();
       ResetQMC = ResetMethod();
       FreeQMC = FreeMethod();
       break;
     ```

6. In `makefile`: Add a new object file to the list of all object files. Then add the rule for the object. It will probably look like this:

   ```
   method.o: method.c method.h qmc.h
           ${CC} -c method.c -o method.o
   ```

7 Randomizing QMC Methods

The purpose of this section is to provide the details for randomizing the QMC methods in this library. In the program shell of Section 3 there are two functions that are used to randomize the point set. Here we give their description.

```
double *GenRandom (void);
```

For all randomizations methods except the `Scrambling` method, this function generates a random vector of the same dimension as was specified in the input file that was read by `LoadInputQMC()`. If the vector was generated successfully then it returns a pointer to it. If there was a problem generating the vector then `NULL` is returned. `NULL` is also returned when `Scrambling` is used.

```
double *AppRandom (double *point, double *r);
```

This function applies to `point` the randomization specified in the input file using the vector `r`. It returns a pointer to the randomized point. If `NULL` is returned then either `InitQMC()` was not called or one of the parameters does not exist.

8 Random Dimension

This section explains how to write a RandQMC program in which the function to be integrated has a random dimension. We remind the user that the only methods for which this

feature is available are Monte Carlo, Korobov rule or polynomial Korobov rule. Also, if a randomization method is used, it should be either the addition of a shift modulo 1, or a digital shift in base 2. The way this library handles random dimension is as follows:

- fix a bound d that should be larger than the dimension most of the time;
- generate the points in d dimensions as usual;
- anything higher than d is computed on-line by the generator.

The third step is achieved through a function called `RPoint(j)` that returns the j^{th} coordinate of the point and is used throughout the program instead of accessing `rpoint[j]`. Hence, in the program shell of Section 3 only the user-defined function has to be modified. More precisely, the function `RPoint()` is defined as

```
double RPoint (int currdim);
```

> This function will return the `currdim` dimension from the point generated. Basically, instead of indexing the array `rpoint`, we call the function `RPoint()` passing it the index, as an argument.

The part of the program shell that we change will look like,

```
for (j = 0; j < GetQMC(QMC_NPOINTS); j++)
   {
     point = QMC();

     rpoint = AppRandom (point, rvector);

     result = function (...); // a user defined function

     StatUpdate1 (stats, INTLOOP, result);
   }
```

The only difference is that `rpoint` is not passed to `function()`. The way that `function()` changes is by replacing `rpoint[i]` by `RPoint(i)`; an example where this is used can be found in the function `RuinREG()` used for the ruin problem in Appendix A, page 49. Note that only the two Korobov methods and Monte Carlo support random dimension right now.

9 Statistical Tools

This section describes how to use the `statsqmc` library to perform basic statistical tasks. This part of the `RandQMC` library is independent from the rest of the library, i.e., it does not use any other part of the `RandQMC` library. Most of the functions available in the `statsqmc` library were inspired from the `STAT` module in the SIMOD simulation package [19].

9.1 A Shell Program

A program shell that uses the statistical tools of the `RandQMC` library consists of at least one `STATSQMC` data type and three parts. The first part is for initializing the `STATSQMC` data type with a call to `InitStat()`. The second part is for updating the statistical data, and retrieving averages, variances and such. The last part is for freeing the `STATSQMC` data type from memory. Before getting into the details, here is an example of a shell that uses these statistical tools.

```
/* A shell for statsqmc */
#include "qmc.h"

#define NUMBLOCKS1 10
#define NUMBLOCKS2 5

{
  STATSQMC stats1 = NULL;
  STATSQMC stats2 = NULL;

  // code WITHOUT any statsqmc function calls

  stats1 = InitStat (NUMBLOCKS1);
  stats2 = InitStat (NUMBLOCKS2);

  // code WITH Stats function calls

  FreeStat (stats1);
  FreeStat (stats2);

  // code WITHOUT any statsqmc function calls
}
```

The `STATSQMC` data type consists of a fixed number of statistical data blocks. In the shell above, `stats1` contains ten blocks and `stats2` contains five blocks. These blocks can be updated with statistical data for either one or two random variables. The sample average, variance and a confidence interval for the mean can be computed for each of these random variables. If two random variables were updated then the sample covariance and correlation can also be computed. The next section describes the functions that update and compute the quantities we just mentioned.

9.2 The Functions

`STATSQMC InitStat (int numblocks);`

> This function initializes a `STATSQMC` data type with the number of blocks equal to `numblocks`. The number of blocks must be greater than 0, and if it is not then `NULL` is returned. A `STATSQMC` type is returned on success.

```
void StatUpdate1 (STATSQMC stats, int block, double entry1);
void StatUpdate2 (STATSQMC stats, int block, double entry1, double entry2);
```

> These functions update the statistical data block at index `block` of the `STATSQMC` data type `stats`. The first block of `stats` is always indexed at 0, and not at 1. The function

`StatUpdate1` is to be used when only one random variable is required. The entry to update with is the parameter `entry1`. The function `StatUpdate2` is to be used when two random variables are required. The parameters `entry1` and `entry2` are the two observations to update with.

```
double StatAverage (STATSQMC stats, int block, int randvar);
```

This function returns the average of the sample in the statistical data block at index `block`. The parameter `randvar` is the random variable to return the average of. It must be equal to one of the following preprocessor defines, `RANDVARONE` or `RANDVARTWO`. Setting `randvar` to `RANDVARONE` will return the average of only the first random variable in the statistical data block. Setting it to `RANDVARTWO` will return the average of the second random variable. If an error occurs then the function will return 0.0, and print an error message. The first possibility for error is if `randvar` was not assigned one of the two preprocessor defines, or was assigned to `RANDVARTWO` but the block was updated with `StatUpdate1`. The second possibility is if `stats` does not exist or if `block` does not satisfy $0 \leq$ `block` $<$ `numblocks`.

```
double StatVariance (STATSQMC stats, int block, int randvar);
```

This function behaves exactly like `StatAverage()`, but instead of returning the average of the block it returns the sample variance.

```
CONFIDENCEINTERVAL StatConfIntval (STATSQMC stats, int block, int randvar,
                                   double level);
```

This function returns the confidence interval for the statistical data block at index `block`. The type, `CONFIDENCEINTERVAL`, that is returned is defined as follows:

```
typedef struct
{
  double low;
  double high;
} CONFIDENCEINTERVAL;
```

The parameter `level` is the confidence level, and must satisfy $0 <$ `level` ≤ 1. For the first three parameters the function behaves exactly like `StatAverage()`. If `level` $= 0$ the function returns 0.0 and prints an error message.

```
double StatCovariance (STATSQMC stats, int block);
```

This function returns the sample covariance of the two random variables in the statistical data block at index `block`. If only one random variable was updated the result of this function is incorrect. If the parameter `stats` does not exist or if `block` is out of range then 0.0 is returned, and an error message is printed.

```
double StatCorrelation (STATSQMC stats, int block);
```

This function behaves exactly like `StatCovariance`, but it returns the sample correlation of the two random variables instead of the covariance.

```
void ResetStat (STATSQMC stats, int block);
```

This function resets the statistical data block at index `block` unless `stats` does not exist or `block` is out of range. In this case, nothing is done.

```
void FreeStat (STATSQMC stats);
```

This function must be called when `stats` is no longer needed. It frees all the blocks contained in `stats` from memory.

9.3 An Example

We now provide an example of a program that uses the statistical tools of the library. It uses two statistical blocks with two random variables each. See Appendix A for additional examples.

```
#include "qmc.h"

#define NUMBLOCKS 2
#define INTLOOP 0
#define EXTLOOP 1

int main (void)
{
  int i, j;
  double val1
  double val2;
  CONFIDENCEINTERVAL cf;
  STATSQMC stats = NULL;

  if ((stats = InitStat (NUMBLOCKS) == NULL))
    return (0);

  for (i = 0; i < 20; i++)
  {
    ResetStat (stats, INTLOOP);
    for (j = 0; j < 1024; j++)
    {
      val1 = (i+1) / (j+1);
      val2 = (j+1) / (i+1);
      StatUpdate2 (stats, INTLOOP, val1, val2);
    }

    val1 = StatAverage (stats, INTLOOP, RANDVARONE);
    val2 = StatAverage (stats, INTLOOP, RANDVARTWO);
    StatUpdate2 (stats, EXTLOOP, val1, val2);
  }

  printf ("Average1: %f", StatAverage (stats, EXTLOOP, RANDVARONE));
  printf ("Variance1: %f", StatVariance (stats, EXTLOOP, RANDVARONE));
  cf = StatConfIntval (stats, EXTLOOP, RANDVARONE, 0.95);
  printf ("ConfidenceInterval1: %f to %f", cf.low, cf.high);

  printf ("Average2: %f", StatAverage (stats, EXTLOOP, RANDVARTWO));
  printf ("Variance2: %f", StatVariance (stats, EXTLOOP, RANDVARTWO));
  cf = StatConfIntval (stats, EXTLOOP, RANDVARTWO, 0.95);
  printf ("ConfidenceInterval2: %f to %f", cf.low, cf.high);
```

```
  printf ("Covariance: %f", StatCovariance (stats, EXTLOOP));
  printf ("Correlation: %f", StatCorrelation (stats, EXTLOOP));

  FreeStat (stats);

  return (1);
}
```

10 Acknowledgments

This work was supported by a starter grant from the University of Calgary through the University Research Grants Committee (URGC), an equipment grant provided by the REE and ACCESS programs at the University of Calgary, and NSERC grant #RGP238959. We also wish to thank Pierre L'Ecuyer for providing us with one of his pseudorandom number generator libraries, which was crucial in the design of the functions and procedures that handle random dimension. We also used different LaTeX macros from Pierre L'Ecuyer to write this guide. Finally, we thank David C. Dembeck and Jia Ye Xiong for having experimented with preliminary versions of the library.

A Examples

This appendix gives two examples that use the `RandQMC` library. The first one considers the evaluation of an Asian option under the Black-Scholes model, and the second one looks at the probability of ruin for an insurance company.

A.1 Asian option problem

We refer the reader to [20] for a more precise description of the problem. Formally, the goal is to estimate the s-dimensional integral:

$$\mu = \int_{[0,1)^s} e^{-rT} \max\left(0, \frac{1}{t}\sum_{i=1}^{s} S(0)\exp\left[(r-\sigma^2/2)t_i + \sigma\sqrt{T/t}\sum_{j=1}^{i}\Phi^{-1}(u_j)\right] - K\right) \mathrm{d}u_1 \ldots \mathrm{d}u_s \tag{5}$$

where

- s is the dimension (number of prices entering the mean);
- r is the risk-free appreciation rate;
- σ is the volatility of the underlying asset

- K is the strike price of the option;
- T is the expiration time of the option;
- t_i is the date at which the ith price of the underlying asset is recorded, for $i = 1 \ldots s$;
- $S(0)$ is the price of the underlying asset at time 0;
- $\Phi(\cdot)$ is the standard normal distribution.

Below is the header file for `AsianOption.c`, which contains the code for a function called `Option` that evaluates the user-defined function f described by (5). This function takes as input the parameters t, r, σ, K, T, $S(0)$, $\vec{u}$, and the date t_1 where the asset's price is recorded for the first time.

```
#ifndef _ASIANOPTION_H_
#define _ASIANOPTION_H_

// ---------------------------------------------------------------------

double Option (double K, double S0, double r, double T1, double T,
               double sigma, int dim, double *point);

// ---------------------------------------------------------------------

#endif
```

Next, we implement the function `Option` in a file called `AsianOption.c`.

```
#include <math.h>
#include "Dist.h"
#include "AsianOption.h"

// ---------------------------------------------------------------------

double Option (double K, double S0, double r, double T1, double T,
               double sigma, int dim, double *point)
{
  int i;
  double S, phi;
  double treal, sum, deltat;
  double temp1, temp2;

  treal = ((double) dim);

  deltat = (T - T1) / treal;

  phi = Normal (0.0, 1.0, point[0]);

  S = exp((r-0.5*sigma*sigma)*(T1+deltat) + (sigma*phi*sqrt(T1+deltat)));

  sum = S;

  temp1 = (r - 0.5 * sigma * sigma) * deltat;
  temp2 = sigma * sqrt(deltat);
```

```
  for(i = 1; i < dim; i++)
    {
      phi = Normal (0.0, 1.0, point[i]);
      S = S * exp (temp1 + temp2 * phi);
      sum = sum + S;
    }

  sum = S0 * (sum / treal) - K;

  if (sum > 0.0)
    return (exp(-r * T) * sum);
  else
    return (0.0);
}

// ------------------------------------------------------------------------
```

`Option` evaluates the function described in (5) at the point `point`. The `Normal()` function returns a standard normal random variable by inverting the normal CDF at `point[i]`. It is included from `Dist.h`, and the implementation is based on [13, pages 95-96]. The function `exp(double x)` evaluates e^x, and is included from `math.h`.

Now, we give the main program for this example. For simplicity, the parameters of the `Option()` function above are hard-coded into the program.

```
#include "qmc.h"
#include "AsianOption.h"

#define NUMBLOCKS 2
#define INTLOOP 0
#define EXTLOOP 1

// ------------------------------------------------------------------------

int main (int argc, char **argv)
{
  int i, j;

  double *point = NULL;
  double *rvector = NULL;
  double *rpoint = NULL;
  double result = 0.0;

  STATSQMC stats = NULL;
  CONFIDENCEINTERVAL cf;

  double K = 100;
  double T1 = 0.246576;  // 90/365
  double S0 = 100.0;
  double r = 0.086177;   // ln 1.09
  double T = 0.328767;   // 120/365
  double sigma = 0.2;

  int n, m, s;

  if(!LoadInputQMC(argv[1]))
```

```
  {
    printf ("Cannot load the input file.");
    return (0);
  }

if (!InitQMC ())
  {
    printf ("Cannot initialize the Quasi Monte Carlo method.");
    return (0);
  }

if ((stats = InitStat (NUMBLOCKS)) == NULL)
  {
    printf ("Cannot initialize stats.");
    FreeQMC();
    return (0);
  }

n = GetQMC(QMC_NPOINTS);
m = GetQMC(QMC_RSHIFTS);
s = GetQMC(QMC_DIMEN);

for (i = 0; i < m; i++)
  {
    ResetStat (stats, INTLOOP);
    ResetQMC();

    rvector = GenRandom();

    for (j = 0; j < n; j++)
      {
        if ((point = QMC()) == NULL)
          {
            printf ("Cannot generate point %d.", j);
            FreeQMC();
            FreeStat (stats);
            return (0);
          }

        if ((rpoint = AppRandom (point, rvector)) == NULL)
          {
            printf ("Cannot apply the randomization.");
            FreeQMC();
            FreeStat (stats);
            return (0);
          }

        result = Option (K, S0, r, T1, T, sigma, s, rpoint);

        StatUpdate1 (stats, INTLOOP, result);
      }

    result = StatAverage (stats, INTLOOP, RANDVARONE);
    StatUpdate1 (stats, EXTLOOP, result);
  }

printf ("Average: %f", StatAverage (stats, EXTLOOP, RANDVARONE));
printf ("Variance: %f", StatVariance (stats, EXTLOOP, RANDVARONE));
```

```
  cf = StatConfIntval (stats, EXTLOOP, RANDVARONE, 0.95);
  printf ("Confidence Interval (95%%): %f to %f", cf.low, cf.high);

  FreeQMC();
  FreeStat (stats);
  return (1);
}

// ------------------------------------------------------------------
```

Below is a makefile that can be used to correctly generate the executable file for this program. It assumes that the main program is called `evaloption.c`, and can be run using `make` or `make -f filename` if it is named `filename` rather than `makefile`.

```
CC = gcc
OBJECTS = AsianOption.o Dist.o

all: ${OBJECTS} MAKEINPUT
        ${CC} ${OBJECTS} evaloption.c -lm -L. -lqmc -o evaloption

#this rule makes the object files for the QMC executable

AsianOption.o: AsianOption.c AsianOption.h Dist.h
        ${CC} -c AsianOption.c -o AsianOption.o

Dist.o:  Dist.c Dist.h
        ${CC} -c Dist.c -o Dist.o

#this rule makes the MAKEINPUT executable

MAKEINPUT: makeinput.c
        ${CC} makeinput.c -lm -L. -lqmc -o makeinput

#this rule cleans up the directory

clean:
        rm -f *.o core *~ AsianOption makeinput
```

Note that the files required to compile the program are `Dist.*, AsianOption.*, evaloption.c, makeinput.c, qmc.h`, and the RandQMC library `libqmc.a`.

To execute the program, type:

```
[qmc] ./evaloption input.dat
```

where `input.dat` is the input file for the parameter list that describes which randomized QMC method is used. This file can be created by running the `makeinput` program with the command

```
 ./makeinput input.dat
```

which will prompt the user as described in Section 5.

A.2 Probability of ruin

For a description of the problem, we refer the reader to [29]. The approach used is regenerative simulation, as discussed in [50, 25]. With this approach, we need to simulate a process (dual of the surplus process) X whose value after the jth claim arrival is given by [29]

$$X_j = \max\left[0, Y_j, X_{j-1}e^{-\delta T_j} - c\left(\frac{1-e^{-\delta T_j}}{\delta}\right) + Y_j\right],$$

where T_j is the interarrival time between the $(j-1)$th and jth claim, Y_j is the amount of the jth claim, c is the premium rate, and δ is the interest rate. In our program, we assume the T_j's are i.i.d. exponential with mean $1/\lambda$, and the Y_j's are i.i.d. exponential with mean μ. Assuming $X(0) = u$, the regeneration points are defined as the times t at which $X(t) = u$.

The estimator based on n cycles is given by

$$\frac{\sum_{i=1}^{n} Z_i/n}{\sum_{i=1}^{n} T_i/n}, \tag{6}$$

where T_i is the length (in time) of the ith cycle, and Z_i is the length of time that the process X has spent below level u in the ith cycle. Since the number of claims in each cycle is random, we need to use a method that supports random dimension.

Below is the header `ruinREG.h` file for our function `RuinREG`.

```
#ifndef _RUINREG_H
#define _RUINREG_H

// ---------------------------------------------------------------------

void RuinREG (double lambda, double mu, double c, double delta, double u,
              double *D, double *sumTi);

// ---------------------------------------------------------------------

#endif
```

The function takes the parameters that are required for the simulation, but notice that the pointer to the randomized point is not passed. More details on this feature are given below, after the description of the file `ruinREG.c` that implements the function `RuinREG`.

```
#include "qmc.h"
#include "ruin.h"
#include "Dist.h"

// ---------------------------------------------------------------------

void RuinREG (double lambda, double mu, double c, double delta, double u,
              double *D, double *sumTi)
{
  int i;
  int done;
  double T, Y, r, interm;
```

```
  double X[2];

  *D = 0.0;
  *sumTi = 0.0;

  X[0] = u;
  done = 0;
  i = 0;

  while (!done)
  {
    T = Expon (1.0 / lambda, RPoint(i));
    Y = Expon (mu, RPoint (++i));

    *sumTi = *sumTi + T;

    interm = X[0] * exp(-delta*T)-c*((1.0-exp(-delta*T))/delta)+Y;

    if (Y > 0.0)
    {
      if (Y > interm)
        X[1] = Y;
      else
        X[1] = interm;
    }
    else if (interm > 0.0)
      X[1] = interm;
    else
      X[1] = 0.0;

    r = (log((X[0]+c/delta)/(u+c/delta)))/delta;

    if (X[0] > u)
    {
      if (T-r > 0.0)
      {
        *sumTi = *sumTi - (T-r);
        done = 1;
      }
    }
    else
      *D = *D + T;

    X[0] = X[1];
    i++;
  }

  *D = *sumTi - *D;

  return;
}

// ---------------------------------------------------------------------
```

The generation of the two random variables `T` and `Y` is done at the beginning of the `while` loop. The function `Expon()` is used to generate `T` and `Y` by inversion, and is included from `Dist.h`. The function `RPoint()` is included from `qmc.h`, and returns the ith coordinate of

the randomized point when called with parameter i. This is used instead of indexing the variable `rpoint` as in the previous example.

Below is the main program `SimRuinReg.c` that generates i.i.d. copies of the estimator (6); each estimator is based on the n points of a randomized QMC point set.

```
#include "qmc.h"
#include "ruin.h"

#define NUMBLOCKS 2
#define INTLOOP 0
#define EXTLOOP 1

int ReadUserInput (char *filename, double lambda, double mu, double c,
                   double delta, double u);

// ---------------------------------------------------------------------

int main(int argc, char **argv)
{
  int i, j;

  double *point = NULL;
  double *rvector = NULL;
  double *rpoint = NULL;

  STATSQMC stats = NULL;
  CONFIDENCEINTERVAL cf;

  // user variables for estimating the probability of ruin
  double lambda, mu, c, delta, u;
  double WBarre, TBarre, cov, mu_hat, sigmav2, ectype;
  double D, sumTi;
  int n, m;

  // check for correct number of arguments
  if(argc != 3)
    {
      printf ("No input file(s) Specified");
      printf ("Usage: %s filename1 filename2", argv[0]);
      printf ("filename1 = QMC parameters input file");
      printf ("Please run MAKEINPUT or specify filename");
      printf ("filename2 = ruinREG parameters input file");
      return (0);
    }

  if(!LoadInputQMC(argv[1]))
    {
       printf ("Cannot load the input file.");
       return (0);
    }

  if (!ReadUserInput (*(argv+2), &lambda, &mu, &c, &delta, &u))
    {
      printf ("Cannot read the user input file.");
      return (0);
```

```
    }

  if (!InitQMC ())
    {
      printf ("Cannot initialize the Quasi Monte Carlo method.");
      return (0);
    }


  if ((stats = InitStat (NUMBLOCKS)) == NULL)
    {
      printf ("Cannot initialize stats.");
      FreeQMC();
      return (0);
    }

  n = GetQMC(QMC_NPOINTS);
  m = GetQMC(QMC_RSHIFTS);

  for (i = 0; i < m; i++)
    {
      ResetStat (stats, Wl_Tl);
      ResetQMC();

      rvector = GenRandom();

      for (j = 0; j < n; j++)
        {
          if ((point = QMC()) == NULL)
            {
              printf ("Cannot generate point %d.", j);
              FreeQMC();
              FreeStat (stats);
              return (0);
            }

          if ((rpoint = AppRandom (point, rvector)) == NULL)
            {
              printf ("Cannot apply the randomization.");
              FreeQMC();
              FreeStat (stats);
              return (0);
            }

          RuinREG (lambda, mu, c, delta, u, &D, &sumTi);
          StatUpdate2 (stats, INTLOOP, D, sumTi);
        }

      D = StatAverage (stats, INTLOOP, RANDVARONE);
      sumTi = StatAverage (stats, INTLOOP, RANDVARTWO);
      StatUpdate2 (stats, EXTLOOP, D, sumTi);
    }

  WBarre = StatAverage (stats, EXTLOOP, RANDVARONE);
  TBarre = StatAverage (stats, EXTLOOP, RANDVARTWO);
  mu_hat = WBarre / TBarre;

  printf ("Average: %f", mu_hat);
```

```
  cov = StatCovariance (stats, EXTLOOP);

  sigmav2 = StatVariance (stats, EXTLOOP, RANDVARONE) +
    (mu_hat * mu_hat * StatVariance (stats, EXTLOOP, RANDVARTWO)) -
    (2.0 * mu_hat * cov);

  ectype = sqrt (sigmav2 / ((double)plist[PRANDSHIFTS)) / TBarre;

  printf ("Standard Deviation: %f", ectype);

  FreeQMC();
  FreeStat (stats);
  return (1);
}

// ------------------------------------------------------------------------

int ReadUserInput (char *filename, double lambda, double mu, double c,
                   double delta, double u)
{
  int i;
  FILE *infile;

  if (!filename)
    return (0);

  if ((infile = fopen (filename, "r")) == NULL)
    return (0);

  fscanf (infile, "%f", (double)lambda);
  fscanf (infile, "%f", (double)mu);
  fscanf (infile, "%f", (double)c);
  fscanf (infile, "%f", (double)delta);
  fscanf (infile, "%f", (double)u);

  fclose (infile);
  return (1);
}

// ------------------------------------------------------------------------
```

Here is the `makefile` that should be used to compile the program `SimRuinReg.c` given above.

```
CC = gcc
OBJECTS = ruinREG.o Dist.o

all: ${OBJECTS} MAKEINPUT
${CC} ${OBJECTS} SimRuinReg.c -lm -L. -lqmc -o SimRuinReg

#this is all the rules to make all the object files for the QMC exectuable

ruinREG.o: ruinREG.c ruinREG.h Dist.h qmc.h
${CC} -c ruinREG.c -o ruinREG.o

Dist.o:  Dist.c Dist.h
${CC} -c Dist.c -o Dist.o

#this is the rule to make the MAKEINPUT executable

MAKEINPUT: makeinput.c
${CC} makeinput.c -lm -L. -lqmc -o makeinput

#This is a rule to clean up the directory

clean:
rm -f *.o core *~ ruinREG makeinput
```

The procedure to run the makefile is similar to what we described in the previous example.

To execute the program `SimRuinReg`, type:

```
[qmc] ./SimRuinReg input.dat userin.dat
```

where `input.dat` is the input file for the parameter list, and `userin.dat` is the user input file for the parameters of the `RuinREG()` function.

References

[1] I.A. Antonov and V.M. Saleev. An economic method of computing LP_τ-sequences. *Zh. Vychisl. Mat. i. Mat. Fiz.*, 19:243–245, 1979. In Russian.

[2] P. Bratley and B. L. Fox. Algorithm 659: Implementing Sobol's quasirandom sequence generator. *ACM Transactions on Mathematical Software*, 14(1):88–100, 1988.

[3] P. Bratley, B. L. Fox, and H. Niederreiter. Algorithm 738: Programs to generate Niederreiter's low-discrepancy sequences. *ACM Transactions on Mathematical Software*, 20:494–495, 1994.

[4] J. Cheng and M.J. Druzdzel. Computational investigation of low-discrepancy sequences in simulation algorithms for bayesian networks. In *Uncertainty in Artificial Intelligence Proceedings 2000*, pages 72–81, 2000.

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

[6] R. Couture, P. L'Ecuyer, and S. Tezuka. On the distribution of k-dimensional vectors for simple and combined Tausworthe sequences. *Mathematics of Computation*, 60(202):749–761, S11–S16, 1993.

[7] H. Faure. Discrépance des suites associées à un système de numération. *Acta Arithmetica*, 61:337–351, 1982.

[8] H. Faure. Variations on $(0, s)$-sequences. *Journal of Complexity*, 17:741–753, 2001.

[9] I. Friedel and A. Keller. Fast generation of. In K.-T. Fang, F. J. Hickernell, and H. Niederreiter, editors, *Monte Carlo and Quasi-Monte Carlo Methods 2000*, pages 257–273. Springer, 2001.

[10] J. H. Halton. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numerische Mathematik*, 2:84–90, 1960.

[11] P. Hellekalek. On the assessment of random and quasi-random point sets. In P. Hellekalek and G. Larcher, editors, *Random and Quasi-Random Point Sets*, volume 138 of *Lecture Notes in Statistics*, pages 49–108. Springer, New York, 1998.

[12] H.S. Hong and F.H. Hickernell. Implementing scrambled digital sequences. Submitted for publication, 2001.

[13] W. J. Kennedy Jr. and J. E. Gentle. *Statistical Computing*. Dekker, New York, 1980.

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

[15] N. M. Korobov. The approximate computation of multiple integrals. *Dokl. Akad. Nauk SSSR*, 124:1207–1210, 1959. In Russian.

[16] P. L'Ecuyer. Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation*, 65(213):203–213, 1996.

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

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

[19] P. L'Ecuyer. *SIMOD: Définition fonctionnelle et guide d'utilisation*, Jan. 2000.

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

[21] P. L'Ecuyer and C. Lemieux. A survey of randomized quasi-Monte Carlo methods. In P. L'Ecuyer M. Dror and F. Szidarovszki, editors, *Modeling Uncertainty: An Examination of Stochastic Theory, Methods, and Applications*. Kluwer Academic Publishers, 2002. 419–474.

[22] P. L'Ecuyer and F. Panneton. A new class of linear feedback shift register generators. In J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, editors, *Proceedings of the 2000 Winter Simulation Conference*, pages 690–696, Pistacaway, NJ, Dec 2000. IEEE Press.

[23] P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An objected-oriented random-number package with many long streams and substreams. *Operations Research*, 50:1073–1075, 2002.

[24] C. Lemieux. *L'utilisation de règles de réseau en simulation comme technique de réduction de la variance*. PhD thesis, Université de Montréal, May 2000.

[25] C. Lemieux and P. L'Ecuyer. Lattice rules for the simulation of ruin problems. In *Proceedings of the 1999 European Simulation Multiconference*, volume 2, pages 533–537, Ghent, Belgium, 1999. The Society for Computer Simulation.

[26] C. Lemieux and P. L'Ecuyer. Randomized polynomial lattice rules for multivariate integration and simulation. To appear in SIAM Journal on Scientific Computing, 2003.

[27] J. Matoušek. On the L_2-discrepancy for anchored boxes. *Journal of Complexity*, 14:527–556, 1998.

[28] M. D. Mckay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21:239–245, 1979.

[29] F. Michaud. Estimating the probability of ruin for variable premiums by simulation. *Astin Bulletin*, 26:93–105, 1996.

[30] H. Morohosi and M. Fushimi. A practical approach to the error estimation of quasi-Monte Carlo integration. Technical Report METR 98-10, The University of Tokyo, Dept. of Math. Engineering and Information Physics, 1998.

[31] H. Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*, volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, Philadelphia, 1992.

[32] H. Niederreiter and C. Xing. The algebraic-geometry approach to low-discrepancy sequences. In P. Hellekalek, G. Larcher, H. Niederreiter, and P. Zinterhof, editors, *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, volume 127 of *Lecture Notes in Statistics*, pages 139–160, New York, 1997. Springer-Verlag.

[33] A. B. Owen. Randomly permuted (t, m, s)-nets and (t, s)-sequences. In H. Niederreiter and P. J.-S. Shiue, editors, *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, number 106 in Lecture Notes in Statistics, pages 299–317. Springer-Verlag, 1995.

[34] A. B. Owen. Latin supercube sampling for very high-dimensional simulations. *ACM Transactions of Modeling and Computer Simulation*, 8(1):71–102, 1998.

[35] F. Panneton. Générateurs pseudo-aléatoires utilisant des récurrences linéaires modulo 2. Master's thesis, Université de Montréal, 2000.

[36] S. Paskov and J. Traub. Faster valuation of financial derivatives. *Journal of Portfolio Management*, 22:113–120, 1995.

[37] G. Pirsic and W. Ch. Schmid. Calculation of the quality parameter of digital nets and application to their construction. *J. Complexity*, 17:827–839, 2001.

[38] W. Ch. Schmid. Shift-nets: a new class of binary digital (t, m, s)-nets. In P. Hellekalek, G. Larcher, H. Niederreiter, and P. Zinterhof, editors, *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, volume 127 of *Lecture Notes in Statistics*, pages 369–381, New York, 1997. Springer-Verlag.

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

[40] I. M. Sobol'. The distribution of points in a cube and the approximate evaluation of integrals. *U.S.S.R. Comput. Math. and Math. Phys.*, 7:86–112, 1967.

[41] I. M. Sobol'. Uniformly distributed sequences with an additional uniform property. *USSR Comput. Math. Math. Phys. Academy of Sciences*, 16:236–242, 1976.

[42] I. M. Sobol' and Yu. L. Levitan. The production of points uniformly distributed in a multidimensional. Technical Report Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976. In Russian.

[43] J. Struckmeier. Fast generation of low-discrepancy sequences. *Journal of Computational and Applied Methematics*, 91:29–41, 1995.

[44] K. S. Tan and P.P. Boyle. Applications of randomized low discrepancy sequences to the valuation of complex securities. *Journal of Economic Dynamics and Control*, 24:1747–1782, 2000.

[45] S. Tezuka. *Uniform Random Numbers: Theory and Practice.* Kluwer Academic Publishers, Norwell, Mass., 1995.

[46] S. Tezuka and P. L'Ecuyer. Efficient and portable combined Tausworthe random number generators. *ACM Transactions on Modeling and Computer Simulation*, 1(2):99–112, 1991.

[47] S. Tezuka and T. Tokuyama. A note on polynomial arithmetic analogue of Halton sequences. *ACM Transactions on Modeling and Computer Simulation*, 4:279–284, 1994.

[48] J. P. R. Tootill, W. D. Robinson, and D. J. Eagle. An asymptotically random Tausworthe sequence. *Journal of the ACM*, 20:469–481, 1973.

[49] B. Tuffin. On the use of low-discrepancy sequences in Monte Carlo methods. Technical Report No. 1060, I.R.I.S.A., Rennes, France, 1996.

[50] F. J. Vázquez-Abad. Rpa pathwise derivative estimation of ruin probabilities. *Insurance: Mathematics and Economics*, 26:269–288, 2000.