Fast Generation of Randomized Low-Discrepancy Point Sets

Ilja Friedel[1] and Alexander Keller[2]

[1] ilja@cs.caltech.edu, Computer Science Dept., California Institute of Technology, Pasadena, CA 91125, USA
[2] keller@informatik.uni-kl.de, Computer Science Dept., University of Kaiserslautern, D-67653 Kaiserslautern, Germany

Abstract. We introduce two novel techniques for speeding up the generation of digital (t, s)-sequences. Based on these results a new algorithm for the construction of Owen's randomly permuted $(t, s)-$sequences is developed and analyzed. An implementation is available at `http://www.mcqmc.org/Software.html`.

1 Introduction

The numerical methods of Monte Carlo and quasi-Monte Carlo integration (see e.g. [Nie92]) approximate integrals

$$If := \int_{[0,1)^s} f(x)dx \approx \frac{1}{N}\sum_{i=0}^{N-1} f(X_i) =: I_N f \tag{1}$$

over the s-dimensional unit cube $[0,1)^s$ by averaging N observations of the integrand f taken at the sample points $X := \{X_0, X_1, \ldots, X_{N-1}\} \subset [0,1)^s$. The Monte Carlo method uses independent identically distributed random sample points acquiring an integration error $|If - I_N f| \in \mathcal{O}\left(N^{-1/2}\right)$ with high probability. The variance of $I_N f$ can be estimated empirically using the function samples of the quadrature. Given an integrand f of bounded variation in the sense of Hardy and Krause, constructing a set X of low-discrepancy sample points allows the quasi-Monte Carlo method to obtain an improved convergence rate of $|If - I_N f| \in \mathcal{O}\left(N^{-1}\ln^s N\right)$. However, it is not known how to estimate the approximation error efficiently from the function samples. The computation of the multiplicative constant of the order is significantly harder than computing the approximation (1) itself.

1.1 Randomized Replications

By using randomized replications of low-discrepancy sample points, we can both exploit the increased convergence properties of quasi-Monte Carlo integration and use the statistical Monte Carlo estimate of variance for error prediction [Owe98b]. For this purpose we require the randomized replications $X_k := \{X_{k,0}, \ldots, X_{k,n-1}\}$, $1 \le k \le r$, of the low-discrepancy points $A := \{A_0, \ldots, A_{n-1}\}$ of a quasi-Monte Carlo quadrature rule to fulfil

- **Property 1: Uniformity.** For fixed $k : X_{k,i} \sim U[0,1)^s$, meaning that the $X_{k,i}$ are uniformly distributed over the unit cube.
- **Property 2: Equidistribution.** $X_1, \dots, X_r$ are low-discrepancy point sets with probability one.

Then we can rewrite the approximation (1) as

$$I_{r,n} f := \frac{1}{r} \sum_{k=1}^{r} \frac{1}{n} \sum_{i=0}^{n-1} f(X_{k,i}) \, , \tag{2}$$

and for $r > 1$ estimate its variance by

$$\mathsf{Var}(I_{r,n} f) \approx \frac{1}{r(r-1)} \sum_{k=1}^{r} \left(\frac{1}{n} \sum_{i=0}^{n-1} f(X_{k,i}) - I_{r,n} f \right)^2 \, , \tag{3}$$

using a total of $N = rn$ samples, where in the limit cases we either have pure quasi-Monte Carlo integration, where no variance estimate is available, or pure Monte Carlo integration. Obviously the ability to estimate the error is paid by sacrificing some of the convergence of the quasi-Monte Carlo integration. Consequently the number r of replications should be chosen just large enough to allow for a sufficiently accurate variance estimate and small enough to preserve the higher accuracy of quasi-Monte Carlo integration.

Cranley-Patterson rotations [CP76] are a simple way to produce random replicates. Owen [Owe95] introduced an even more powerful method for randomly scrambling (t, m, s)-nets and (t, s)-sequences. In the following we consider the fast construction and randomization of these low-discrepancy points.

2 (t, m, s)-Nets and (t, s)-Sequences

We briefly recall the construction of digital (t, m, s)-nets and (t, s)-sequences in the framework as derived by Niederreiter [Nie87] from previous work on special cases by Sobol' [Sob67] and Faure [Fau82].

Definition 1. For a fixed dimension $s \geq 1$ and an integer base $b \geq 2$ the subinterval

$$J = \prod_{j=1}^{s} \left[a_j \cdot b^{-d_j}, (a_j + 1) \cdot b^{-d_j} \right) \subseteq [0,1)^s$$

with $0 \leq a_j < b^{d_j}$, $a_j, d_j \in \mathbb{N}_0$, is called an **elementary interval**. For integers $0 \leq t \leq m$ a point set A of b^m points in $[0,1)^s$ is called a $(\mathbf{t}, \mathbf{m}, \mathbf{s})$**-net in base b**, if every elementary interval of size $\lambda_s(J) = b^{t-m}$ contains exactly b^t points. For an integer $t \geq 0$ a sequence $A_0, A_1, \dots$ of points in $[0,1)^s$ is a $(\mathbf{t}, \mathbf{s})$**-sequence in base b** if, for all integers $k \geq 0$ and $m > t$, the point set consisting of the A_i with $kb^m \leq i < (k+1)b^m$ is a (t, m, s)-net in base b.

2.1 Construction by the Digital Approach

Let

$$[0,1)_{b,M} := \left\{kb^{-M} \mid k = 0, \ldots, b^M - 1\right\} \subset [0,1)$$

be the set of all fixed-point numbers that can be written using M digits in base b. We define the components $A_i^{(j)}$ of a point set $A = \{A_0, \ldots, A_{n-1}\}$ by

$$A_i^{(j)} = \sum_{k=1}^{M} a_{i,k}^{(j)} \cdot b^{-k} =_b 0.a_{i,1}^{(j)} a_{i,2}^{(j)} \ldots a_{i,M}^{(j)} \in [0,1)_{b,M}\,, \text{ where} \tag{4a}$$

$$a_{i,k}^{(j)} := \eta_k^{(j)}\left(s_{i,k}^{(j)}\right), \text{ and } s_{i,k}^{(j)} := \sum_{l=0}^{M-1} c_{k,l}^{(j)} \cdot \psi_l\bigl(d_{i,l}\bigr)\,. \tag{4b}$$

The digit $d_{i,l} \in \mathbb{Z}_b := \{0, \ldots, b-1\}$ is defined to be the l-th digit of the integer

$$i =: \sum_{l=0}^{M-1} d_{i,l} \cdot b^l$$

represented in base b. For a commutative ring $(R, +, \cdot)$ with $|R| = b$ elements for each dimension $1 \le j \le s$

$$C^{(j)} := \left(c_{k,l}^{(j)}\right)_{k=1,l=0}^{M,M-1} \in R^{M \times M}$$

is the generator matrix, and $\eta_k^{(j)} : R \to \mathbb{Z}_b$ and $\psi_l : \mathbb{Z}_b \to R$ are two families of bijections.

If the constructed point set A is a (t,m,s)-net in base b, then it is also called a **digital** (t,m,s)-net **constructed over** R. **Digital** (t,s)**-sequences** are defined analogously. The quality of a digital sequence is mainly determined by the choice of the ring R and the generator matrices $C^{(j)}$.

The commutative ring R in general is implemented using two lookup tables of size b^2 for multiplication and addition, and a lookup table of size b for the additive inverse. Algorithms for computing rings are known. However, most implementations of (t,s)-sequences described in literature restrict themselves to fields, which can only be generated, if the number of elements $b = p^q$ is a power of a prime p. In the simple case of $q = 1$, the ring R is equal to $\mathbb{F}_p := \mathbb{Z}/p\mathbb{Z}$. Polynomial rings $R[X]$ over R are frequently used for the construction of the generator matrices $C^{(j)}$[BFN92,Nie92] and should therefore be supported by an implementation. Good generator matrices were proposed by Sobol' [Sob67], Faure [Fau82], Niederreiter and others. Further details can be found in [Nie92,Sch95] and in most other papers in the references.

Identifying the elements of R and $\mathbb{Z}_b$ by some canonical bijection $\nu : \mathbb{Z}_b \to R$, we define the permutations $\tilde{\psi}_l$ and $\tilde{\eta}_k^{(j)}$ by $\psi_l =: \tilde{\psi}_l \circ \nu$ and

$\eta_k^{(j)} =: \tilde{\eta}_k^{(j)} \circ \nu^{-1}$. This is done because permutations are easier to handle in an implementation. In many papers these permutations are chosen as identity, making implementations a bit shorter and faster. Other authors suggest searching for 'good' permutations or to fix some randomly chosen permutations in the initialization phase of the program. This is also a method for creating randomized replications as defined in Sect. 1.1.

2.2 Fast Generation of (t, s)-Sequences

For given ring R, generator matrices $C^{(j)}$, and permutations ψ_l and $\eta_k^{(j)}$, it is simple but very time consuming to compute the digital sequence using (4b) and (4a). The calculation of n vectors of an s-dimensional sequence with M digits accuracy requires $\Theta(nsM^2)$ elementary operations. Small bases, especially for $b = 2, \ldots, 10$, lead to large M, resulting in a very slow calculation.

- **Gray Code Counter.** Antonov and Saleev [AS79] suggested the use of a Gray code counter in base b for incrementing i. Thus exactly one digit $d_{i,l}$ of i changes with every counter increment. In consequence only one summand in (4b) has to be recomputed. This reduces the number of elementary operations to $\Theta(nsM)$. While using the Gray code generates the points in a rearranged order, it does not affect their property of being a $(t, s)-$sequence in base b.
- **Standard b-ary Counter.** A traditional counter in base b in every step changes its first (rightmost) digit $d_{i,0}$, in every b-th step its second digit $d_{i,1}$, in every b^2-th step its third digit $d_{i,2}$ and so on. On the average there are not more than

$$1 \leq 1 + b^{-1} + b^{-2} + \ldots + b^{-M+1} \leq \sum_{l=0}^{\infty} b^{-l} \leq 2$$

  changing digits per increment. If the sum $s_{i,k}^{(j)}$ in (4b) is stored in memory, it is possible to calculate the difference of $a_{i+1,k}^{(j)}$ and $a_{i,k}^{(j)}$ in $\mathcal{O}(2)$ elementary operations. The resulting algorithm runs in $\Theta(nsM)$ time and for small bases is only a bit slower than the Gray code variant. However, the sequence is generated in the 'correct' order and it is not necessary to implement the Gray code counter. For the efficient generation of the Halton sequence an example is found in [HW64].

Based on these observations we introduce two new acceleration techniques for the fast computation of (t, s)-sequences.

Sum Splitting and Buffering. The first idea is to reduce the average number of increments of the standard b-ary counter from $\sum_{l=0}^{\infty} b^{-l}$ to $\sum_{l=L}^{\infty} b^{-l}$

with $L \geq 0$. The sum in (4b) can be split into two parts, namely

$$s_{i,k}^{(j)} = \sum_{l=0}^{M-1} c_{k,l}^{(j)} \psi_l\big(d_{i,l}\big) = \sum_{l=L}^{M-1} c_{k,l}^{(j)} \psi_l\big(d_{i,l}\big) + \underbrace{\sum_{l=0}^{L-1} c_{k,l}^{(j)} \psi_l\big(d_{i,l}\big)}_{=:r_{i,k}^{(j)}} .$$

For arbitrary $h \in \mathbb{N}$ we have $d_{i,l} = d_{i+h\cdot b^L,l}$ for $0 \leq l < L$ and in consequence $r_{i,k}^{(j)} = r_{i+b^L,k}^{(j)}$. By tabulating the first $0 \leq i < b^L$ sums $s_{i,k}^{(j)}$ for each digit $1 \leq k \leq M$ and each dimension $1 \leq j \leq s$, we can efficiently compute the next b^L values

$$s_{i+b^L,k}^{(j)} = s_{i,k}^{(j)} + \Delta ,$$

where

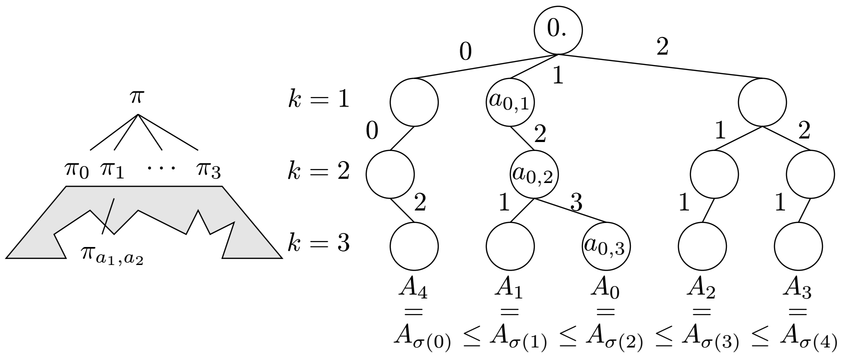
$$\Delta := \sum_{l=L}^{M-1} c_{k,l}^{(j)} \psi_l\big(d_{i+b^L,l}\big) - \sum_{l=L}^{M-1} c_{k,l}^{(j)} \psi_l\big(d_{i,l}\big) = \sum_{l=L}^{M-1} c_{k,l}^{(j)} \cdot \Big(\psi_l\big(d_{i+b^L,l}\big) - \psi_l\big(d_{i,l}\big)\Big)$$

has to be determined every b^L steps only. A further optimization is possible by checking whether Δ is zero. In many cases digits don't change for quite a long time and a single `if`-statement statistically prevents - for Niederreiter sequences in about 2/3 of all cases - updates of whole columns of the table. In practice the length b^L of the table should be chosen approximately between 30 and 100.

Vectorizing the Ring. The second concept reduces the number of summands by artificially increasing the size of the digits $a_{i,k}^{(j)}$. An implementation should provide a vector space-like structure R^q with b^q elements. If R^q supports vector addition $u + v$ and $-u$ with $u, v \in R^q$, and a scalar-vector multiplication $x \cdot v$ with $x \in R, v \in R^q$, then (4b) and (4a) can be computed with q elements in parallel. In this case the variable k is counting $1, q + 1, 2q + 1, 3q + 1, \ldots$

Because the lookup tables for the vectorized operations are growing fast with b and q, this technique can only be used for small bases $2 \leq b \leq 16$. Fortunately these are the bases where the calculation of the $(t, s)-$sequence is still slow.

Specialization to Base $b = 2$. Digital sequences in base $b = 2$ allow to introduce many simplifications. $\mathbb{F}_2 = (\{0, 1\}, +, \cdot)$ is the only commutative ring with two elements. Vectorized addition and multiplication efficiently can be realized using the bitwise `XOR` and `AND`. Since the symmetric group S_2 consists only of two permutations, *identity* and *transpose*, it is possible to emulate these permutations by `XOR`ing with either 0 or 1. Today's computers work in the binary system and offer $q = 32$ or $q = 64$ bit vectors thus providing the operations needed for vectorizing R as described in the previous section without the need of using very space consuming tables.



π
π_0 π_1 $\cdots$ π_3
π_{a_1,a_2}
$k=1$
$k=2$
$k=3$
0.
$a_{0,1}$
$a_{0,2}$
$a_{0,3}$
$A_4 = A_{\sigma(0)} \le A_1 = A_{\sigma(1)} \le A_0 = A_{\sigma(2)} \le A_2 = A_{\sigma(3)} \le A_3 = A_{\sigma(4)}$

Fig. 1. On the left the permutation tree $\tau = (\pi, \pi_0, \pi_1, \pi_2, \pi_3, \pi_{0,0}, \ldots, \pi_{3,3})$ of depth $M = 3$ and base $b = 4$ is sketched. On the right we see an instance of a sequence tree for $A = \{A_0 =_4 0.123, A_1 =_4 0.121, A_2 =_4 0.211, A_3 =_4 0.221, A_4 =_4 0.002\}$ with $n = |A| = 5$.

3 Randomized (t, s)-Sequences

Looking at the digital approach for the construction of $(t, m, s)-$nets, the ring R, and the families of bijections ψ_l and $\eta_k^{(j)}$ are subject to randomization in order to obtain variance estimates by replications as introduced in Sect. 1.1. While choosing random rings is not very practicable, scrambling the permutations leads to meaningful results [Fri98,Mat98]. Owen presented [Owe95] and analyzed [Owe97a] a more general random replication scheme that fulfils the two required properties of uniformity and equidistribution which are needed for the variance estimation (3).

Definition 2. Let A be a (t, m, s)-net or a (t, s)-sequence in base b. A **scrambled replicate X of A** is obtained by computing the components $X_i^{(j)} =_b 0.x_{i,1}^{(j)} x_{i,2}^{(j)} x_{i,3}^{(j)} \cdots x_{i,M}^{(j)}$ where

$$
\begin{aligned}
x_{i,1}^{(j)} &:= \pi^{(j)}\left(a_{i,1}^{(j)}\right) \\
x_{i,2}^{(j)} &:= \pi^{(j)}_{a_{i,1}^{(j)}}\left(a_{i,2}^{(j)}\right) \\
&\vdots \\
x_{i,M}^{(j)} &:= \pi^{(j)}_{a_{i,1}^{(j)}, a_{i,2}^{(j)}, \ldots, a_{i,M-1}^{(j)}}\left(a_{i,M}^{(j)}\right),
\end{aligned}
\qquad (5)
$$

and every permutation $\pi^{(j)}$ has been randomly drawn with uniform probability from the symmetric group S_b of all permutations over the set $\{0, \ldots, b-1\}$. While the permutations are mutually independent, each permutation depends on the $k-1$ leading digits of $A_i^{(j)}$.

Following this definition, an equidistributed, independently chosen **permutation tree** (see Fig. 1)

$$\tau^{(j)} := \left(\pi^{(j)}, \pi_0^{(j)}, \pi_1^{(j)}, \ldots, \pi_{b-1}^{(j)}, \pi_{0,0}^{(j)}, \ldots, \pi_{b-1,b-1}^{(j)}, \ldots, \pi_{b-1,\ldots,b-1}^{(j)}\right)$$
$$\in \Sigma_{b,M} := S_b \times S_b^b \times S_b^{b^2} \times S_b^{b^3} \times \cdots \times S_b^{b^{M-1}} \cong S_b^{\frac{b^M-1}{b-1}}$$

has to be constructed for every dimension $1 \le j \le s$. Here $\Sigma_{b,M}$ denotes the set of all permutation trees. Now the randomization scheme as defined by (5) can be regarded as a mapping $\tau := \left(\tau^{(1)}, \ldots, \tau^{(s)}\right)$ with

$$\begin{aligned} \tau^{(j)} : \mathit{Seq}_{b,M} &\to \mathit{Seq}_{b,M} \\ A^{(j)} &\mapsto X^{(j)}, \end{aligned}$$

where

$$\mathit{Seq}_{b,M} := \{A : \mathbb{N}_0 \to [0,1)_{b,M}\}$$

denotes the set of all one-dimensional sequences A over $[0,1)_{b,M}$. This mapping is injective, because the inverse $a_{i,k}^{(j)}$ of any $x_{i,k}^{(j)}$ is found by recursively computing $a_{i,k}^{(j)} = \left(\pi^{(j)}_{a_{i,1}^{(j)},\ldots,a_{i,k-1}^{(j)}}\right)^{-1}\left(x_{i,k}^{(j)}\right)$. Since the set $[0,1)_{b,M}$ is finite and $\tau^{(j)}$ is defined for every element of this set, $\tau^{(j)}$ is even bijective.

3.1 Construction of Lazy Random Permutations

We briefly recall the generation of random permutations (see e.g. [Knu81]). Let the array `P[0..b-1]` contain b different symbols in any order.

```
FOR i=0 TO b-2 DO exchange P[i] and P[i+$U\{0,\ldots,b-i-1\}$]
```

generates a random (independent, equidistributed over S_b) permutation of these symbols, where $U\{0,\ldots,k\}$ denotes a random element drawn with the uniform distribution over the set $\{0,\ldots,k\}$. As a corollary to this algorithm we note that the concatenation of an arbitrary permutation σ with a random, equidistributed permutation π_ω, will result in a random, equidistributed permutation $\pi_\omega \circ \sigma$.

In some cases we might need a sequence $(\sigma^{(k)})_{k=1}^{\infty}$ of random injective functions $\sigma^{(k)} : \{0,\ldots,l_k-1\} \to \{0,\ldots,b-1\}$ with a priori unknown $l_k \le b$. It is possible to use random permutations $\pi_\omega^{(k)}$ for this task. But if we have $l_k \ll b$ in many cases, then this is very expensive. In order to avoid the problem of not knowing l_k in advance, we set $l_1 := b$. Then $\sigma^{(1)} : \{0,\ldots,b-1\} \to \{0,\ldots,b-1\}$ is a random permutation. With this initialization it is possible to construct $\sigma^{(k+1)}$ from $\sigma^{(k)}$ and l_k by

```
FOR i=0 TO $l_k$-1 DO exchange P[i] and P[i+$U\{0,\ldots,b-i-1\}$].
```

The input in P[...] is $\sigma^{(k)}$ and the output is $\sigma^{(k+1)}$. We call this procedure the calculation of partial or **lazy random permutations** [Ben99].

3.2 Fast Generation of Randomized (t, s)-Sequences

For bases $b > 2$ the arising problems are either memory consumption or computation time. Every permutation needs at least b storage cells and one permutation tree $\tau^{(j)}$ consists of $\frac{b^M-1}{b-1}$ permutations. Even moderate parameters like $M = 5$, $b = 31$, $s = 10$ will use a huge amount of memory and computation time, making the direct approach infeasible. One way to avoid this problem is to choose small M. Another option is to create special techniques and data structures for dealing with the permutations.

We want to choose M large compared with the accuracy of machine numbers, e.g. $b^M \approx 2^{32}$ or even $b^M \approx 2^{64}$. Each component $A_i^{(j)}$, of a sequence $A^{(j)} \in Seq_{b,M}$ is interpreted as a path through the permutation tree $\tau^{(j)}$ as illustrated in Fig. 1. For each dimension j the union of these paths forms another tree, which we briefly call **sequence tree**.

Efficient Traversal of Permutation Trees. Sequence trees are sparse for short sequences, where $n \ll b^M$. This leads to the conclusion that for efficient construction of the randomized sequence it is important to cut all unused (with respect to the sequence tree) random permutations in $\tau^{(j)}$. This can be done by observing the following property: Reading the leaves of the sequence tree from left to right returns the elements of $A^{(j)}$ in ascending order (see Fig. 1).

For the implementation we first sort the sequence $A^{(j)}$ and store the permutation σ^{-1} that undoes the sorting. Then the leaves of the permutation tree are traversed from left to right and the digits of $A_{\sigma(i)}^{(j)}$ are scrambled by (5). Note that the permutation tree won't materialize at any time, since for every component $A_{\sigma(i)}^{(j)}$ only the M permutations along the path from the root to the leaf have to be kept in memory. After scrambling the digits of all numbers in the sequence, the algorithm will undo the initial sorting using σ^{-1} to restore the original order of $A^{(j)}$.

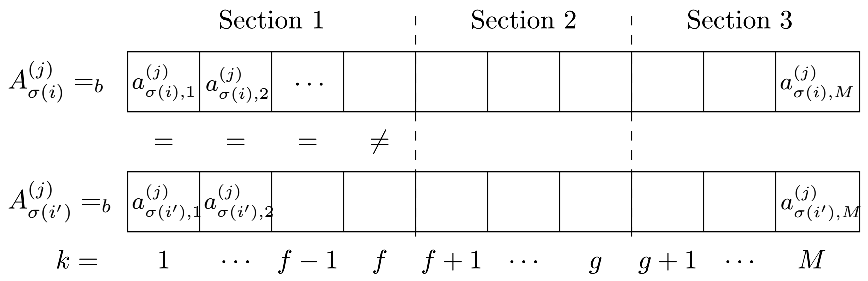
Using a quicksort[1] for sorting the sequence $A^{(j)}$, the permutation σ^{-1} is obtained in $\mathcal{O}(n \log n)$ steps on the average, or with some extra memory by radix sort in $\Theta(n)$ steps. The algorithm needs to store the sequence A, the permutation σ^{-1}, one scrambled component $X^{(j)}$, and M permutations out of S_b. If enough memory is available, the algorithm will work fine for long sequences and arbitrary bases, especially with radix sort. However, if many random realizations of a short sequence are required, the sparse regions of the tree become big and the multiplicative constant $c(M, b)$ of the order $\mathcal{O}(nsbM)$ will slow down the performance.

The advantage of the new algorithm is that it will work with moderate memory requirements and a very simple data structure. The length n of the

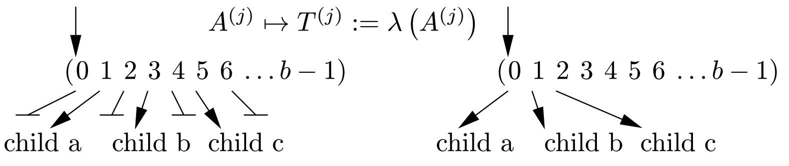
[1] A randomized version of quicksort will guarantee this time stochastically, but it is often slower than the unrandomized algorithm. Sometimes the digital net might have one sorted component. Then traditional quicksort is not a good choice.

sequence, however, must be determined before construction. The bookkeeping of the M permutations by backtracking along the paths of the sequence tree is still an expensive operation. But choosing new random permutations, especially for large b, is the most expensive task of this algorithm: A full permutation is created for every node of the sequence tree, even in its sparse regions. The improved algorithm in the next section deals with this problem elegantly.

Lazy Generation of Permutation Trees. After sorting the j-th component of the sequence A as in the previous algorithm, at position i we have the component $A^{(j)}_{\sigma(i)}$. Looking ahead to the first different component $A^{(j)}_{\sigma(i')}$ at position $i' := \min\{\hat{\imath} > i | A^{(j)}_{\sigma(i)} \neq A^{(j)}_{\sigma(\hat{\imath})}\}$, the digits of $A^{(j)}_{\sigma(i)}$ can be divided into three sections:



- *Digit section 1* is determined by the shared leading digits including the first differing digit $a^{(j)}_{\sigma(i),f} \neq a^{(j)}_{\sigma(i'),f}$. For these digits the permutations remain the same.
- *Digit section 2:* For all digits at positions $f+1, \ldots, M$ new permutations have to be chosen. However, by a second look ahead starting from i' we can identify i'' and such determine the number $g-1$ of shared digits with $a^{(j)}_{\sigma(i'),k} = a^{(j)}_{\sigma(i''),k}$ for $k = 1, \ldots, g-1$. If now $f < g$ the new permutations for the digits $k = f+1, \ldots, g$ are used more than once, but probably less than b times. If $f \geq g$ there is no digit section 2 and we skip to digit section 3.
  Instead of scrambling the sequence A directly, the algorithm first transforms A to the sequence T, which is obtained by shifting the children of every node of the sequence tree as far as possible to the left (see also Fig. 2). Note that this does not affect the relative order of the children and ensures that no child will have an unused link to its left.


$A^{(j)} \mapsto T^{(j)} := \lambda\left(A^{(j)}\right)$
$(0\ 1\ 2\ 3\ 4\ 5\ 6\ \ldots b-1)$
child a
child b
child c
$(0\ 1\ 2\ 3\ 4\ 5\ 6\ \ldots b-1)$
child a
child b
child c

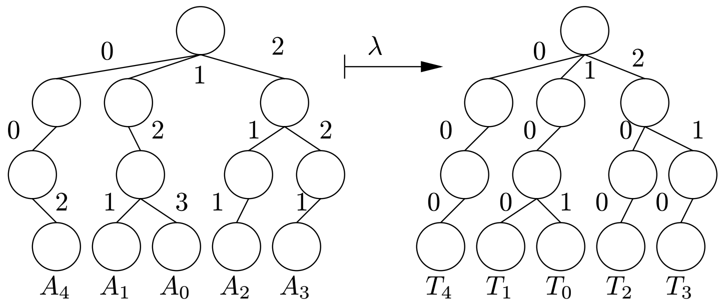


Fig. 2. The leftshift transformation $A^{(j)} \mapsto T^{(j)} := \lambda(A^{(j)})$ applied to the sequence tree from the example in Fig. 1.

Now the permutations that have to be chosen for the digits of section 2 are efficiently realized by lazy permutations as introduced in Sect. 3.1.

- *Digit section 3* is formed by the remaining least significant digits. However looking ahead at the next two differing components $A^{(j)}_{\sigma(i')}$ and $A^{(j)}_{\sigma(i'')}$, we realize, that each of the newly chosen random permutations would be used only once for permuting a single digit. In consequence it is sufficient to draw random digits for $x^{(j)}_{\sigma(i),g+1}, \ldots, x^{(j)}_{\sigma(i),M}$. This is efficiently realized by vectorizing, i.e. drawing only one random number from $U\{0, \ldots, b^{M-g} - 1\}$.

Note that all randomly drawn permutations and digits are independent and equidistributed over the dedicated space.

All interesting information about the sequence $A^{(j)}_{\sigma(i)}$ is stored in the permutations $\left(\sigma^{(j)}\right)^{-1}$ and in the numbers $f^{(j)}_{\sigma(i)}$ that describe where the sequence tree ramifies. Given these two arrays we can quickly compute multiple independent randomizations X of A at roughly twice the memory as compared to the previous algorithm. The leftshift transformation allows to recycle random permutations in the lazy way. As experiments show this reduces the calls to the pseudo random number generator per computed $X^{(j)}_i$ to around 2 to 3 on the average, i.e. to $2sn \ldots 3sn$ for the full sequence X.

Specialization to Base $b = 2$. The actions taken for digit sections 2 and 3 result to be the same in the vectorized approach, since applying a random permutation means `XOR`ing a random bit vector, which in fact cannot be distinguished from using the random bits itself. Consequently $X^{(j)}_{\sigma(i')}$ is computed from $X^{(j)}_{\sigma(i)}$ by `XOR`ing an M-bit vector with $f - 1$ leading zeros followed by a one for changing the branch in the sequence tree, while the remaining bits are chosen at random. Then the number sn of calls to the random number generator is exactly the same as using a plain random sequence.

Correctness. It is not obvious why the leftshift transformation λ of $A^{(j)}$ to $T^{(j)}$ as depicted in Fig. 2 will yield correct results and whether T still is a (t, s)-sequence: In general it is not possible to reconstruct the original sequence A from T. For the clarity of presentation we omit the component superscripts (j) in what follows.

In order to prove the correctness, we need to define two sequences A and B to be equivalent by

$$A \sim B \Leftrightarrow_{def} \exists \tau \in \Sigma_{b,M} : A = \tau(B) .$$

Note that '$\sim$' is an equivalence relation, because $\Sigma_{b,M}$ includes the identity function (reflexivity), permutation trees are closed under concatenation (transitivity), and a unique inverse τ^{-1} for $\tau \in \Sigma_{b,M}$ can be constructed as shown in Sect. 3 (symmetry). The set of equivalence classes of $Seq_{b,M}$ under $\sim$ is denoted by $Seq_{b,M}/\sim$. We write $[A] \in Seq_{b,M}/\sim$ and call every $A \in Seq_{b,M}$ a representative of its equivalence class $[A]$. Now it is possible to reformulate Owen's first proposition:

Proposition 1 (Equidistribution). *If $A \in Seq_{b,M}$ is a $(t, m, s)-$net in base b, then every member of $[A]$ is a $(t, m, s)-$net.*

By previous remarks we know that $[T] = [X]$ and it remains to show that $[A] = [T] = [X]$.

The transformation from A to T is done node by node as illustrated in Fig. 2. Shifting the children to the left is an injective operation. Hence it can be done node-wise with well chosen permutations from S_b. Hence it is possible to construct a $\tau \in \Sigma_{b,M}$ with $T = \tau(A)$. Now it is obvious that T is just a canonical representative of $[A]$. The output

$$X = \rho_\omega(T) = \rho_\omega\big(\tau(A)\big) = (\rho_\omega \circ \tau)(A)$$

of the algorithm is randomly equidistributed over $[A]$, because $\rho_\omega \in \Sigma_{b,M}$ is chosen randomly equidistributed and the concatenation $\rho_\omega \circ \tau$ does not affect this property.

For the implementation rounding errors due to finite precision must be considered. The correctness proof doesn't take such errors into account. If a rounding error cannot be avoided, the affected point should move toward the center of the corresponding elementary interval. If only a single point leaves its corresponding elementary interval, then the convergence rate of the constructed sequence might not be optimal.

Discussion of the new Algorithm. Owen discusses the problem of constructing randomized (t, s)-sequences very briefly in [Owe95]. He remarks that the computation of the scrambling after some simplifications requires $\Omega(nsM)$ storage cells for the permutations and can be done with $\mathcal{O}(nsM)$ calls to the random number generator. One of the suggested simplifications is to choose M such that $b^M \approx n$. This is reasonable for $(0, s)$-sequences, but will bound the maximum length n of the sequence by b^M, which is also true for the algorithms presented in this paper. Otherwise, small errors are introduced.

Matoušek [Mat98] made the proposal to store the permutation tree only up to a certain depth M_1, with $1 \leq M_1 \leq M$. Associated to each leaf of this tree the seed of the random number generator was stored. With these seeds it was possible to recompute each of the b^{M_1} omitted subtrees of depth b^{M-M_1} in case that more information about the subtree was needed. Obviously this method saves some memory, but the random number generator is used more often than necessary.

Comparing these previous attempts with our new algorithm suggests that all are linear in n and s in storage and running time. (There might be a factor of $\log n$ for tree traversal or sorting.) However the multiplicative constant $c(b, M)$ of the new algorithm should be much smaller, because the used data structures are mostly arrays and hence much simpler. Also no result is calculated twice, and the random number generator is used close to a minimum number of times.

4 Numerical Experiments

Similar to [Owe97b,Owe98c] we analyze the empirical distribution and estimate the convergence rate of the randomized $(t, s)-$sequences applied to the numerical integration of

$$f_s\left(x^{(1)}, \ldots, x^{(s)}\right) := 12^{s/2} \cdot \prod_{j=1}^{s} \left(x^{(j)} - 0.5\right) ,$$

where the integral $If_s = 0$ and the variance $\mathbf{Var}(f_s) = 1$. Since digital sequences are used to construct the randomized sequences, they are tested implicitly. Finally we compare the efficiency of all discussed methods. Our `C++` library implementing the presented sequence generators is available under `http://www.mcqmc.org/Software.html`.

4.1 Empirical Distribution

In Fig. 3 we observe the expected Gaussian not only for short, but also for very long-running sequences.

4.2 Convergence Rate

In Fig. 4 we experimentally verify the predicted convergence rate of $\mathcal{O}(N^{-1.5})$ of our new scrambling scheme that should be observed starting from $N \approx s^s$. Then 10 independent realizations of the randomized Niederreiter sequences

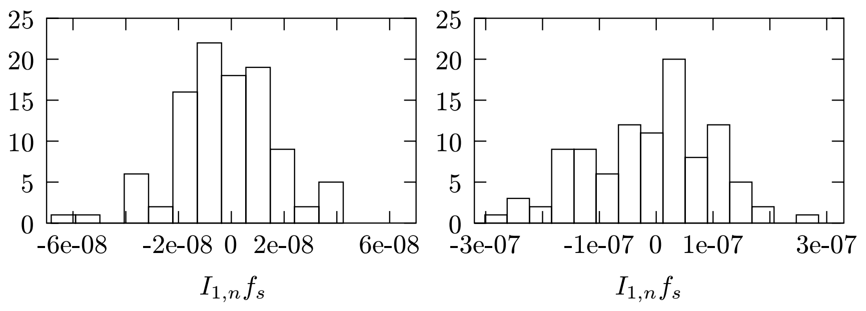


Fig. 3. Frequency of occurence of 100 independent realizations of $I_{1,n}f_s$. The left picture shows the result for scrambled Niederreiter sequences with $b = 2, s = 4, N = n = 2^{24} = 16777216$ and the right one with $b = 11, s = 4, N = n = 11^7 = 19487171$.

are drawn and their absolute errors $|If_s - I_{1,n}f_s|$ are averaged to obtain smooth graphs, which are shown in Fig. 5 for fixed dimension $s = 4$, and in Fig. 6 for $b = s$, where we used $N = 1 \cdot n$.

4.3 Efficiency of the Point Sequence Generators

The sequence created by every generator was of length $N = 1,000,000$, dimension $s = 16$ and at least 32 bits accuracy ($b^M \geq 2^{32}$). In some cases, if memory requirements were too big or extreme running times occurred, then extrapolation methods were used. For instance the fully randomized sequences were tested with $N = 500,000$ and $s = 16$ and the resulting times have been multiplied by two. The results are displayed in Table 1. Note that the memory requirements are only hints, because memory consumption is varying for different bases. The different generators in the table are:

- The default system random number generator `drand48()` is used to provide a lower bound for all fully randomized generators.
- `RandomSequence` in fact is the same as `drand48()`, however the created sequence is stored in memory in order to give an idea about memory access times.
- `LatinHypercube` sampling requires twice as many calls to `drand48()` plus random memory access.
- The `HaltonSequence` as defined by Halton [HW64].
- `DigitalSequence_naive` implements the naive approach of directly computing (4b) and (4a).
- `DigitalSequence_classic.` The technique, where a standard counter is used and only the effect of changing digits is recalculated.
- `DigitalSequence_gray_code.` The technique, where a Gray code counter is used and only the effect of the changing digit is recalculated.
- `DigitalSequence_medium_base` realizes sum splitting with a standard counter and table sizes for $10, \ldots, 100$ vectors of the sequence.

- `DigitalSequence_advanced` extends the techniques of `DigitalSequence_medium_base` by increasing the digit size to about 4 to 6 bits by vectorizing. This makes running times nearly independent of the base b.
- `DigitalSequence_base_2`. This generator is not very fine-tuned and could be optimized with the techniques as described in this article.
- `RandomizedTSSequence` is an implementation of the lazy generation of permutation trees technique. It first calls `DigitalSequence_advanced` in order to create a digital $(t, s)-$sequence (the time in the table includes the time for this operation). Then the sequence is randomized using the lazy generation of permutation trees. The time for `::random_restart()` is measured by executing only phase 2 of this algorithm, which returns an independent randomization of the initial $(t, s)-$sequence.
- `RandomizedTSSequence_base_2` is the specialized version of `RandomizedTSSequence` for base $b = 2$. `DigitalSequence_base_2` is called at the beginning.
- `LatinSupercube(8,8)` pads together two eight-dimensional `RandomizedTSSequence[_base_2]`s as defined by Owen [Owe98a]. The `random_restart()` function call of this generator is faster than creating a `LatinHypercube` point set, because the latter needs $2 \cdot 16 \cdot 10^6$ calls to the random number generator, while this generator just involves $(16 + 2) \cdot 10^6$ calls. As can be seen in Table 1, it is possible to randomize the order of the vectors 'in place'. If the Latin Supercube samples are based on randomized sequences, no memory overhead is introduced.

5 Conclusion

New techniques for speeding up the generation of digital (t, s)-sequences and for realizing Owen's scrambling technique have been presented. There is still potential for optimization by coupling both the digital sequence generation and random number generation more closely, by using more sophisticated sorting techniques, and by reducing the asymptotic memory requirements.

Acknowledgements

The authors are especially thankful to Fred Hickernell and Harald Niederreiter for their comments on the draft version of this paper. The first author wants to thank Peter Schröder for the financial support. The second author was supported by the German Science Foundation DFG.

Table 1. Calculation times for generating $1,000,000$ samples in 16 dimensions with at least 32 bit accuracy. Note the abbreviations `DSeq` for `DigitalSequence` and `Rand` for `Randomized`.

Generator	*No base*	Base b							Memory
	-	2	3	5	13	31	97	257	
`drand48()`	5.3s	-	-	-	-	-	-	-	1 KByte
`RandomSequence`	7.6s	-	-	-	-	-	-	-	130 MBytes
`LatinHypercube`	26s	-	-	-	-	-	-	-	64 MBytes
`HaltonSequence`	1.1s	-	-	-	-	-	-	-	5 KBytes
`DSeq_naive`	-	2320s	800s	350s	153s	97s	55s	39s	10 KBytes
`DSeq_classic`	-	209s	105s	57s	34s	26s	19s	16s	50 KBytes
`DSeq_gray_code`	-	141s	97s	51s	33s	25s	19s	16s	50 KBytes
`DSeq_medium_base`	-	68s	45s	32s	20s	17s	13s	19s	1 MByte
`DSeq_advanced`	-	15s	14s	16s	21s	17s	14s	19s	1 MByte
`DSeq_base_2`	-	2.2s	-	-	-	-	-	-	5 KBytes
`RandTSSequence`	-	132s	125s	104s	96s	84s	82s	86s	150 MBytes
`::random_restart()`	-	56s	38s	38s	34s	32s	28s	32s	
`RandTSSequence_base_2`	-	44s	-	-	-	-	-	-	150 MBytes
`::random_restart()`	-	14s	-	-	-	-	-	-	
`LatinSupercube(8,8)`	-	54s	125s	112s	108s	96s	90s	98s	150 MBytes
`::random_restart()`	-	24s	48s	48s	48s	42s	42s	44s	

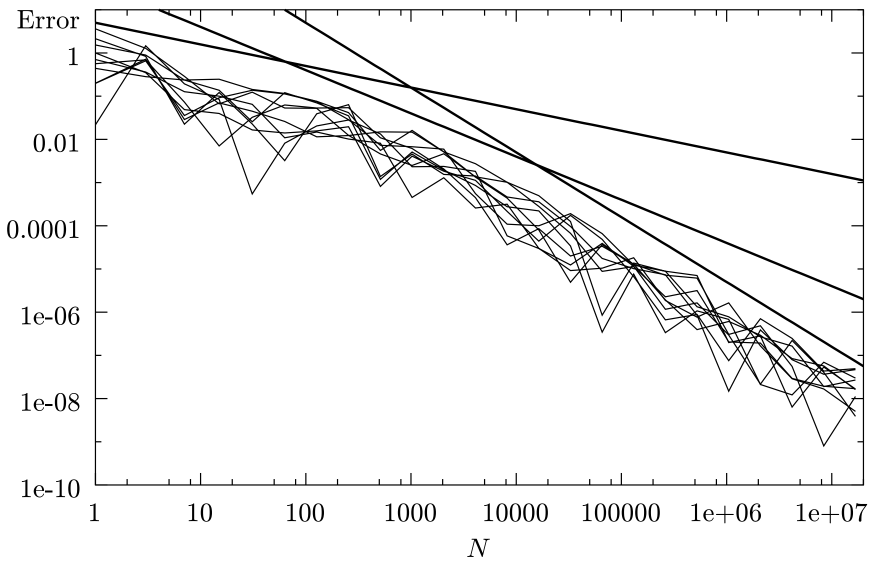


Fig. 4. Absolute error $|If_4 - I_{1,n}f_4|$ of ten realizations of a randomized Niederreiter sequence in base $b = 2$, $N = n = 2^i, 0 \leq i \leq 24$, and $s = 4$. The thick lines indicate the convergence rates of $\mathcal{O}(N^{-0.5})$, $\mathcal{O}(N^{-1})$ and $\mathcal{O}(N^{-1.5})$. Starting at $N \approx 1000$ the convergence is significantly faster than $\mathcal{O}(N^{-1})$, but slightly slower than $\mathcal{O}(N^{-1.5})$.

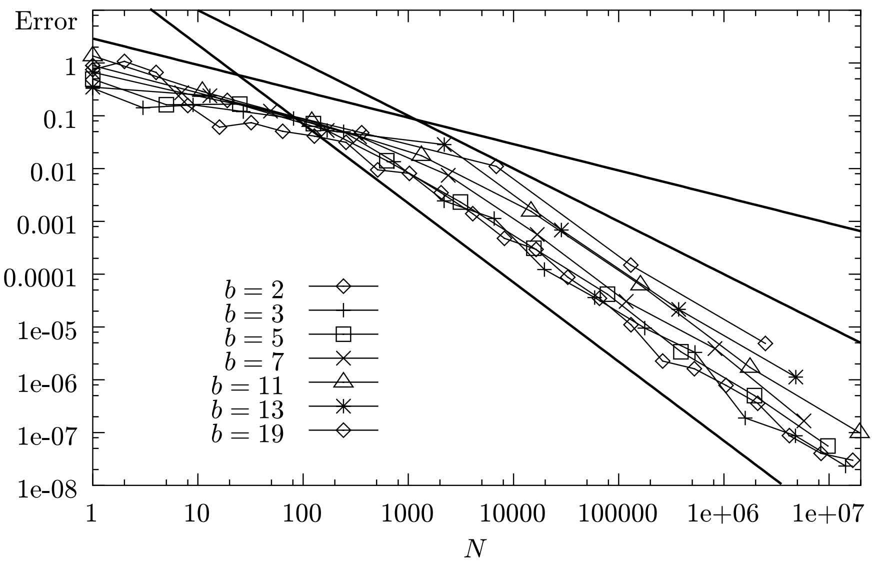


Fig. 5. Averaged absolute error for dimension $s = 4$ and bases $b = 2, 3, 5, 7, 11, 13, 19$. Between $N = 200$ and $N = 8000$ the convergence rate switches to $\mathcal{O}(N^{-1.5})$ for all bases. The sequences with parameter $b = 2$ perform slightly better than most others.

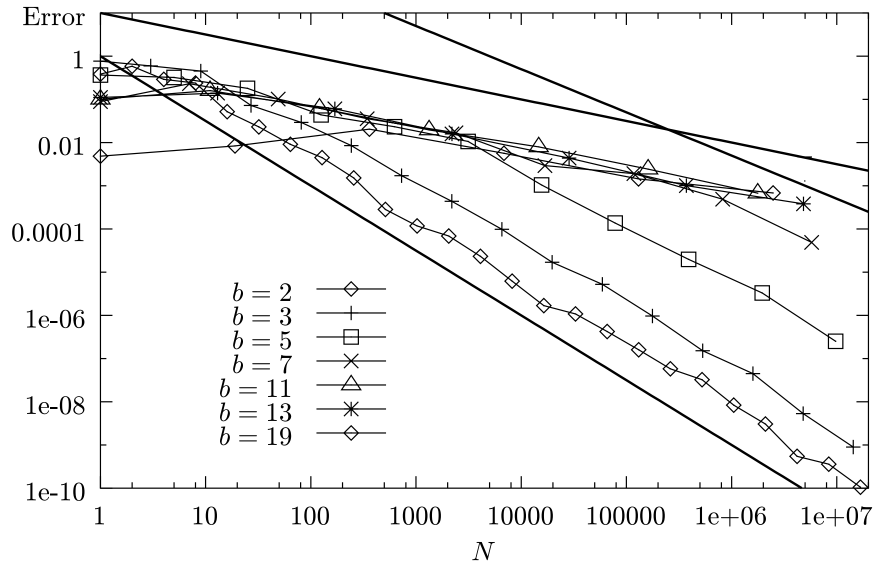


Fig. 6. Averaged absolute error for $s = b = 2, 3, 5, 7, 11, 13, 19$. For all dimensions up to $s = 7$ we observe a switch to the superior convergence rate at about $N \approx s^s$. The behavior of the sequences with $s = b = 19$ and $N = 1$ or $N = 19$ is exceptional. This can be explained with the special form of the integral and the high dimensionality.

References

[AS79] I. Antonov and V. Saleev, *An economic method of computing LP_τ-sequences*, USSR Comput. Math. Math. Phys. **19** (1979), no. 1, 252–256.

[Ben99] J. Bentley, *Programming Pearls*, 2nd ed., Addison Wesley, 1999.

[BFN92] P. Bratley, B. Fox, and H. Niederreiter, *Implementation and tests of low-discrepancy sequences*, ACM Trans. Modeling and Comp. Simulation **2** (1992), no. 3, 195–213.

[CP76] R. Cranley and T. Patterson, *Randomization of number theoretic methods for multiple integration*, SIAM Journal on Numerical Analysis **13** (1976), 904–914.

[Fau82] H. Faure, *Discrépance de suites associées à un système de numération (en dimension s)*, Acta Arith. **41** (1982), no. 4, 337–351.

[Fri98] I. Friedel, *Abtastmethoden für die Monte-Carlo und quasi-Monte-Carlo-Integration*, Universität Kaiserslautern, 1998.

[HW64] J. Halton and G. Weller, *Algorithm 247: Radical-inverse quasi-random point sequence*, Comm. ACM **7** (1964), no. 12, 701–702.

[Knu81] D. Knuth, *The Art of Computer Programming Vol. 2: Seminumerical Algorithms*, Addison Wesley, 1981.

[Mat98] J. Matoušek, *On the L-2-discrepancy for anchored boxes*, J. of Complexity **14** (1998), no. 4, 527–556.

[Nie87] H. Niederreiter, *Point sets and sequences with small discrepancy*, Monatsh. Math. **104** (1987), 273–337.

[Nie92] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods*, SIAM, Pennsylvania, 1992.

[Owe95] A. Owen, *Randomly permuted (t,m,s)-nets and (t,s)-sequences*, Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing (H. Niederreiter and P. Shiue, eds.), Lecture Notes in Statistics, vol. 106, Springer, 1995, pp. 299–315.

[Owe97a] A. Owen, *Monte Carlo variance of scrambled net quadrature*, SIAM J. on Numerical Analysis **34** (1997), no. 5, 1884–1910.

[Owe97b] A. Owen, *Scrambled net variance for integrals of smooth functions*, Annals of Statistics **25** (1997), no. 4, 1541–1562.

[Owe98a] A. Owen, *Latin supercube sampling for very high dimensional simulations*, ACM Trans. Modeling and Comp. Simulation **8** (1998), 71–102.

[Owe98b] A. Owen, *Monte Carlo extension of quasi-Monte Carlo*, Winter Simulation Conference, IEEE Press, 1998.

[Owe98c] A. Owen, *Scrambling Sobol and Niederreiter-Xing points*, J. of Complexity **14** (1998), no. 4, 466–489.

[Sch95] W. Schmid, *(t,m,s)-Nets: Digital Construction and Combinatorial Aspects*, Ph.D. thesis, Universität Salzburg, 1995.

[Sob67] I. Sobol', *On the distribution of points in a cube and the approximate evaluation of integrals*, Zh. vychisl. Mat. mat. Fiz. **7** (1967), no. 4, 784–802.