Implementation and Tests of Low-Discrepancy Sequences

PAUL BRATLEY
Université de Montréal

BENNETT L. FOX
University of Colorado
and
HARALD NIEDERREITER
Austrian Academy of Sciences

Low-discrepancy sequences are used for numerical integration, in simulation, and in related applications. Techniques for producing such sequences have been proposed by, among others, Halton, Sobol', Faure, and Niederreiter. Niederreiter's sequences have the best theoretical asymptotic properties. The paper describes two ways to implement the latter sequences on a computer and discusses the results obtained in various practical tests on particular integrals.

Categories and Subject Descriptors: G.1.4 [**Numerical Analysis**]: Quadrature and Numerical Differentiation; I.6 [**Computing Methodologies**]: Simulation and Modeling

General Terms: Algorithms

Additional Key Words and Phrases: Low-discrepancy sequences, Niederreiter sequences, Quasi-Monte Carlo methods, quasirandom sequences

1. INTRODUCTION

Low-discrepancy sequences are useful in quasi-Monte Carlo methods for numerical integration, and in simulation and optimization, as well as being interesting in their own right. Surveys of the applications of low-discrepancy sequences can be found in Hua and Wang [9] and Niederreiter [11]. Several methods for producing such sequences have been proposed by Halton [7],

Authors' addresses: P. Bratley, Département d'informatique et de r.o., Université de Montréal, C.P. 6128, Succursale A, Montréal (Québec), Canada H3C 3J7. email: bratley@iro.umontreal.ca; B. L. Fox, Department of Mathematics, University of Colorado, Campus Box 170, P.O. Box 173364, Denver, CO 80217-3364, USA. email: bfox@cudenver.bitnet; H. Niederreiter, Institute for Information Processing, Austrian Academy of Sciences, Sonnenfelsgasse 19, A-1010 Vienna, Austria. email: nied@qiinfo.oeaw.ac.at.

Sobol' [15], Faure [5], Niederreiter [13], and others.

Fox [6] compares the efficiency in practice of sequence generators implementing the proposals of Halton and Faure, as well as of a pseudorandom sequence; Bratley and Fox [3] extend this comparison to the technique suggested by Sobol'. Davenport and Cheng [4] attempt an implementation of the method proposed by Niederreiter [13], but their work contains errors in the underlying finite-field arithmetic and in the implementation of certain linear recursions.

In this paper we describe two ways of implementing Niederreiter's low-discrepancy sequences, and we report practical tests with multidimensional numerical integrations using these sequences.

Following Niederreiter [13], we recall the definition of discrepancy. For N points $x_1, \ldots, x_N$ in the s-dimensional half-open unit cube $I^s = [0, 1)^s$, $s \geq 1$, and a subinterval J of I^s, we put

$$D(J; N) = A(J; N) - V(J)N,$$

where $A(J; N)$ is the number of n, $1 \leq n \leq N$, with $x_n \in J$ and $V(J)$ is the volume of J. Then the *discrepancy* $\Delta(N)$ of the points $x_1, \ldots, x_N$ is defined by

$$\Delta(N) = \sup_J |D(J; N)|,$$

where the supremum is extended over all half-open subintervals $J = \prod_{i=1}^{s}[0, u_i)$ of I^s. For a sequence $x_1, x_2, \ldots$ of points in I^s we define $\Delta(N)$ to be the discrepancy of the first N terms of the sequence. It is believed, but not yet proved, that the best possible bound for the discrepancy is of the form

$$\Delta(N) \leq C_s(\log N)^s + O\big((\log N)^{s-1}\big) \qquad \text{for all} \quad N \geq 2.$$

Thus, the principal theoretical aim in the construction of low-discrepancy sequences is to find sequences of points in I^s with a bound of this form where the constant C_s is as small as possible.

The techniques proposed by Niederreiter [13] yield sequences with the smallest values of C_s currently known. However, it is not certain that sequences with asymptotically good behavior will necessarily perform well in practical applications, where only a finite number of points near the beginning of the sequence will be used. Nor is it obvious that every sequence with good theoretical properties can be implemented efficiently on a computer. In this paper we therefore describe ways to improve the distribution of initial segments of these sequences and detail two possible ways of implementing Niederreiter's method. We give some measures of their efficiency in practice, as well as comparisons with generators derived from other techniques.

The general principles of Niederreiter's method of generating quasirandom sequences are outlined in the following section. Section 3 has some remarks about this generator. Our implementations are described in Section 4. Section 5 presents some results obtained using these programs. Finally, Section 6 summarizes our experience and our recommendations concerning quasirandom generators in general and Niederreiter's generator in particular.

2. NIEDERREITER'S GENERATOR

2.1 General Considerations

A complete account of the theoretical basis of Niederreiter's method of generating quasirandom sequences can be found in [13]. Here we give only an informal description of the necessary computations, including the simplifications and minor modifications we found desirable.

Niederreiter shows how to generate sequences in any integer base $b \geq 2$. Simplifying drastically, what this means is the following. Define an *elementary interval* in base b as an interval of the form

$$E = \prod_{i=1}^{s} \left[\frac{a_i}{b^{d_i}}, \frac{a_i + 1}{b^{d_i}} \right)$$

with integers $d_i \geq 0$ and integers $0 \leq a_i < b^{d_i}$ for $1 \leq i \leq s$. An elementary interval E is thus a subinterval of the unit cube I^s whose ith side has length $1/b^{d_i}$. Informally, it is one of the pieces obtained when we dissect I^s into b parts by parallel cuts equally spaced along one of the axes, then further dissect these pieces in the same way, and so on. Any such elementary interval E has volume $1/b^d$ for some integer $d \geq 0$.

A quasirandom sequence in base b generated by Niederreiter's method has the property that strings of b^m successive points divide themselves evenly among certain elementary intervals in base b. Slightly more formally, let $x_1, x_2, \ldots$ be an s-dimensional sequence and let $t \geq 0$ be an integer. Then for all integers $k \geq 0$ and $m > t$, exactly b^t of the b^m points x_n, $kb^m < n \leq (k + 1)b^m$, fall in every elementary interval in base b with volume b^{t-m}. A sequence with this property is called a (t, s)-*sequence* in base b. It is easy to appreciate intuitively why this property leads to sequences with low discrepancy.

As remarked above, Niederreiter shows how to generate such sequences in any base $b \geq 2$. However, the construction when b is a prime or the power of a prime is considerably simpler than the construction for general b. Since there is no compelling practical reason to implement the general case (according to Niederreiter [13], up to dimension 20 the minimal C_s is always found among the cases we treat), we confined ourselves to those bases b which are primes or prime powers. In these cases, of course, there exists a finite field F_b of order b, and much of the construction involves operations in this field. When we refer to prime powers in the sequel, we include primes.

On the other hand, there does exist a compelling practical reason to treat the case $b = 2$ specially. The binary nature of computers means that the field elements of F_2 can be represented by bits, so simultaneous arithmetic operations in F_2 can be carried out very efficiently when—as is usual—the computer has full-word, bitwise *and* and *exclusive-or* instructions. By taking advantage of the latter instruction we were able to write a program which runs many times faster than the general prime-power case, although it implements essentially the same construction. The differences between the base-2 program and the program for prime-power bases are discussed in the following section.

Suppose therefore from now on that the base b is a prime power. Let $B = \{0, 1, \ldots, b-1\}$ be the set of digits in base b. Niederreiter's construction requires us to choose [13, page 54]:

(1) a commutative ring M with identity and $|M| = b$;
(2) bijections ψ_r and λ_{ij} from B to M and from M to B, respectively, with certain stated properties; and
(3) elements $c_{jr}^{(i)} \in M$, again with stated properties.

For M, our implementation takes F_b, the finite field with b elements. When b is prime, this poses no problem: arithmetic in F_b is integer arithmetic modulo b, whose programming is straightforward. When b is not a prime, however, life is harder. Since the values of b we use in practice are small (b must be less than 50 in the current implementation), we chose to solve the problem by reading precalculated addition and multiplication tables for F_b from a file, and doing all field arithmetic by table look-up. To avoid cluttering the program with code for special cases, we do this even when b is prime, except that in this case there is no need to precalculate the tables. For more details, see Appendix A.

Doing arithmetic by table look-up means that the field elements can be labelled arbitrarily, except for zero and the identity, and hence the choice of ψ_r and λ_{ij} is trivial. We merely let the digits $0, 1, \ldots, b-1$ correspond to field elements labelled $0, 1, \ldots, b-1$ and vice versa. The auxiliary program which produces the field-arithmetic tables is free to interpret these labels (except 0 and 1) in any way it chooses. Thus, we shall suppose from now on that ADD(x, y) gives the result of adding the elements x and y of F_b, and that MUL and SUB are defined similarly. This result can be interpreted as the label of a field element or as a digit in base b as required by the context; similarly, x and y may be either labels or digits. In other words, the bijections ψ_r and λ_{ij} can simply be omitted.

The calculation of the elements $c_{jr}^{(i)}$ is less straightforward, so we postpone discussing this. For the moment, suppose these elements are available.

2.2 The One-Dimensional Case

For clarity, suppose further for the time being that we are working in just one dimension, so that we can suppress all mention of the coordinate index i used by Niederreiter [13] (most often as a superscript, but occasionally as a subscript). Our aim is thus to generate a sequence of values $x_1, x_2, \ldots$, $0 \le x_n < 1$, with low discrepancy over the unit interval.

To generate x_n, we need the representation of $n-1$ as an integer in base b, using digits $a_i \in B$. For convenience of implementation we chose not to allow more than R base-b digits, where R is the largest integer such that $b^R - 1$ is representable as a FORTRAN integer on our computer. This means of course that we cannot generate more than b^R values of a quasirandom sequence in base b, but this limitation is of little practical concern. Similarly, the value of x_n is generated as a fraction in base b, and for the same reason we decided to limit this fraction to R figures. This decision too has only minor

practical consequences. Unless double precision is used, real variables in FORTRAN typically have less precision than integers, so in any case extra precision would be lost. For double-precision work, it would be easy to extend the generator to produce, say, $2R$ figures, with a consequent loss of speed on most (but not all) computers.

With these restrictions and the conventions concerning field arithmetic described above, Niederreiter's method may be summarized as follows. To generate x_n, let

$$n - 1 = a_{R-1}a_{R-2}\ldots a_1a_0, \quad a_i \in B$$

be the base-b representation of $n - 1$. Then x_n is given as a base-b fraction by

$$x_n = 0.d_1d_2\ldots d_R, \quad d_i \in B$$

where

$$d_j = \operatorname*{ADD}_{r=0}^{R-1} \mathrm{MUL}(c_{jr}, a_r), \quad 1 \le j \le R.$$

(Compare this to [13, page 54].) In practice we calculate an integer Q_n whose base-b representation is

$$Q_n = d_1d_2\ldots d_R$$

and then take $x_n = Q_n/b^R$.

2.3 Calculating c

We continue to suppose that we are working in one dimension, suppressing all mention of coordinate subscripts or superscripts i.

Niederreiter defines the elements c of F_b in terms of other elements a [13, p. 56, Eq. (7)]. Although a formal definition of these elements a appears just above this in Eq. (6), for our purposes they are defined on [13, p. 65] where Niederreiter shows how the general definition can be made operational by a convenient choice of parameters. Near the foot of that page, we find a definition of the elements a in terms of yet further elements v. Finally, working backwards up the page, these elements v are calculated from the coefficients b_i of a certain polynomial $p(x)^j$.

Note that the elements v thus have *two* implicit parameters that were suppressed in [13] for simplicity. Besides an implicit parameter i for the coordinate of interest, which we have also suppressed, they have a further implicit parameter j, the power of $p(x)$ used to calculate them.

All this leads to the following provisional algorithm for calculating the elements c. Increment i means set $i \leftarrow i + 1$, where i is arbitrary.

(1) Choose a suitable monic polynomial $p(x)$ with coefficients in F_b. Let the degree of $p(x)$ be $e \ge 1$. Set $j \leftarrow 0$, $q \leftarrow -1$, and $u \leftarrow e$.

(2) Increment j. If $u = e$, do step (3); otherwise, go to step (4).

(3) Increment q and set $u \leftarrow 0$. Calculate first

$$b(x) = p(x)^{q+1} = x^m - b_{m-1}x^{m-1} - \cdots - b_0,$$

say, a polynomial of degree $m = e(q + 1)$, and then the elements v_i, $0 \le i \le R + e - 2$, using $v_i = 0$ for $0 \le i \le m - 2$, $v_{m-1} = 1$, and

$$v_i = \operatorname*{ADD}_{k=1}^{m} \mathrm{MUL}(b_{m-k}, v_{i-k}) \quad \text{for } m \le i \le R + e - 2.$$

(4) For $0 \le r \le R - 1$, set $c_{jr} \leftarrow v_{r+u}$. Increment u. If $j < R$, go to step (2); otherwise, stop.

To justify this algorithm, notice that j, q, and u correspond to the same three variables as defined in Eq. (7) of [13, page 56]. Thus

$$c_{jr} = a(q + 1, u, r) = v_{r+u},$$

using the equation near the foot of page 65. Here the elements v have (besides an implicit parameter for the coordinate) the implicit parameter $q + 1$. The reader may verify in the algorithm that whenever q changes, the polynomial $b(x)$ and the values of v are recalculated. In practice, of course, we obtain the new polynomial $b(x)$ by multiplying the old one by $p(x)$, not by calculating $p(x)^{q+1}$ from scratch. We do not compute q explicitly; instead, we update the degree of $b(x)$ each time step (3) above is executed. Section 3.3 of our paper modifies the choice $v_i = 0$, $0 \le i \le m - 2$, for reasons explained there.

Since $c_{jr} = v_{r+u}$, $r \le R - 1$ and $u \le e - 1$, it is certainly sufficient to calculate v_i for $0 \le i \le R + e - 2$. Remember that the coefficients of the polynomials $p(x)$ and $b(x)$ are elements of F_b, not ordinary integers. Thus, $b(x)$ must be calculated using ADD and MUL, not ordinary integer addition and multiplication.

Appendix B gives a small numerical example to illustrate the ideas outlined above.

2.4 The Multidimensional Case

Generalizing this procedure to s dimensions, we generate sequences of quasirandom points or vectors

$$q_n = (x_n^{(1)}, x_n^{(2)}, \ldots, x_n^{(s)}).$$

To do this, it suffices to take a different polynomial $p^{(i)}(x)$, $1 \le i \le s$, for each coordinate and to calculate different $c_{jr}^{(i)}$ and hence different $x_n^{(i)}$ for each coordinate of q_n. The polynomials $p^{(i)}(x)$ must be pairwise coprime, and for optimal results their degrees must be as small as possible [13, p. 58]. This is achieved by listing all monic irreducible polynomials over F_b in a sequence $p^{(1)}(x), p^{(2)}(x), \ldots$ such that $\deg(p^{(h)}(x)) \le \deg(p^{(k)}(x))$ whenever $h \le k$. Then we let $p^{(1)}(x), \ldots, p^{(s)}(x)$ be the first s polynomials in this list. The degree of $p^{(s)}(x)$ grows like $\log s$.

For our implementation, tables of monic irreducible polynomials for each required field F_b are precalculated by an auxiliary program. Like the field-arithmetic tables, they are then read from a file when required. Published

tables of irreducible polynomials are widely available (see for instance [10]) should their use be deemed preferable.

3. REMARKS ON THE GENERATOR

3.1 Specializing the Generator for Base 2

Section 2 outlined the theory underlying the generation of sequences in any prime-power base b. Construction of such sequences involves operations in the field F_b. When $b = 2$, such operations are particularly easy to implement on a computer. The elements of F_2 are bits 0 or 1, multiplication is ordinary integer multiplication (or logical *and*, if one prefers to use lower-level bit operations), and addition and subtraction are both the logical exclusive-or operation. In these conditions, Niederreiter's generator can be implemented very efficiently.

Suppose then that we are working in base $b = 2$, and choose R as the number of bits in an integer variable (not counting the sign). This is consistent with the definition of R used in Section 2.2 above. For clarity we shall, as before, suppress the superscript i used to indicate the coordinate of interest. Exactly the same computation is to be carried out for every coordinate of the quasirandom points we are generating.

Now the elements c_{jr} are single bits. We can therefore pack a number of these elements into an integer variable. Suppose we pack the elements c_{jr}, $1 \le j \le R$, into a single word C_r in such a way that the representation of C_r in binary is

$$C_r = c_{1r}c_{2r}\ldots c_{Rr}.$$

Since MUL in F_2 is the same as ordinary multiplication and since ADD in F_2 is the exclusive-or operation, it is easy to see that

$$Q_n = a_0C_0 \bullet a_1C_1 \bullet \cdots \bullet a_{R-1}C_{R-1},$$

where $\bullet$ denotes a bit-by-bit exclusive-or operation.

This would already yield quite an efficient way of calculating Q_n given the binary representation $a_{R-1}\ldots a_0$ of $n-1$. As in [3], however, we adopt a suggestion of Antonov and Saleev [2] and calculate Q_n from

$$Q_n = g_0C_0 \bullet g_1C_1 \bullet \cdots \bullet g_{R-1}C_{R-1}, \qquad (*)$$

where $g_{R-1}\ldots g_0$ is the Gray-code representation of $n-1$. Antonov and Saleev show that for $k = 1, 2, \ldots$ this shuffles each segment of length 2^k of the sequence being generated, but does not affect the property which in our terminology is that of being a (t, s)-sequence in base 2, and hence does not affect the discrepancy bound.

We remind the reader that the Gray code for m and the Gray code for $m+1$ differ in only one bit position. If a_k is the rightmost zero-bit in the ordinary binary representation of m (add a leading zero to m if there are no others), then g_k is the bit whose value changes. Using this property, and defining Q_n by $(*)$, we can compute Q_{n+1} in terms of Q_n as

$$Q_{n+1} = Q_n \bullet C_r,$$

where b_r is the rightmost zero-bit in the ordinary binary representation of $n - 1$. This calculation is very fast. To start the recurrence, we take $Q_1 = 0$.

We take advantage of this efficient way of calculating Q_n in the base-2 program discussed below.

3.2 Optimal Bases

Niederreiter [12, Theorems 4.2 and 4.3] gives explicit discrepancy bounds of the form

$$\Delta(N) \le U(t, s, b, N) = A(t, s, b)(\log N)^s + O\big((\log N)^{s-1}\big),$$

where the explicit form of the big-O term follows from $U(t, s, b, N)$. These bounds are valid for all (t, s)-sequences in base b. Niederreiter [13] constructs certain $(t(s, b), s)$-sequences in base b, giving a bound of the form

$$\Delta(N) \le B(s, b)(\log N)^s + O\big((\log N)^{s-1}\big).$$

For each s, he chooses a base $b = b(s)$ to minimize $B(s, b)$—giving a bound of the form

$$\Delta(N) \le C_s(\log N)^s + O\big((\log N)^{s-1}\big).$$

His values of C_s are the smallest currently known. The corresponding values of $b(s)$ do not necessarily minimize the actual discrepancy for practical values of N. This may explain why taking $b = 2$ in his sequences gave results ranging from roughly as accurate to far more accurate than for $b = b(s)$ for the values of N and s used in our experiments, for the integrals tried. A further explanation is that the discrepancy bound becomes tighter as b becomes larger. Another possible reason is that the implied constant in the big-O term of the bound grows significantly with b.

Niederreiter shows [13, pp. 63–64] that for sequences in base 2 the constant $B(s, 2)$ obtained using his method is the same as that obtained using Sobol's generator [3, 15] for $1 \le s \le 7$. (However, the sequences generated are not the same.) Further, for $s \ge 8$ the constant $B(s, 2)$ obtained using Niederreiter's method is smaller than that obtained by Sobol'. By itself, this does not prove that Niederreiter's base-2 sequences are better than Sobol's for $s \ge 8$ because $B(s, 2)$ appears in a discrepancy bound rather than in the actual discrepancy. However, that caveat does not apply to the following comparison. It is also shown in [13] that for $s \ge 8$ the Niederreiter sequences are (t, s)-sequences in base 2 with a smaller value of t than the sequences of Sobol', and thus the former sequences have finer uniformity properties than the latter.

Niederreiter further shows that to minimize $B(s, b)$ for fixed s, it is usually better to choose a base b different from 2. His Table 1 [13, p. 63] gives the value $b(s)$ of b which minimizes $B(s, b)$ for $2 \le s \le 20$. Niederreiter notes in particular that the smallest value C_s of $B(s, b)$ is not always obtained using a base $b(s) \ge s$ as in some earlier constructions. Specifically, for dimensions 4, 8, 9, and 14, argmin $B(s, b)$ is not the smallest prime $\ge s$. Faure [5] uses the latter.

3.3 The Leading-Zeros Phenomenon

The elements v_i in Section 2.3 are calculated by a linear recursion of order $m = e(q + 1)$ with initial values $v_i = 0$ for $0 \le i \le m - 2$ and $v_{m-1} = 1$. This choice of initial values produces the phenomenon that the leading digit $d_1 = d_1(n)$ in the base-b representation of x_n is 0 for all $n \le b^{e-1}$. To see this, note that for $j = 1$ we have $c_{1r} = v_r$ for all $r \ge 0$, and so in particular $c_{1r} = 0$ for $0 \le r \le e - 2$. Moreover, for $n \le b^{e-1}$ the digits $a_r = a_r(n)$ in the base-b representation of $n - 1$ satisfy $a_r = 0$ for $r \ge e - 1$, and so

$$d_1(n) = \operatorname*{ADD}_{r=0}^{R-1} \mathrm{MUL}(c_{1r}, a_r(n)) = 0 \quad \text{for } n \le b^{e-1}.$$

For the s-dimensional sequence $q_1, q_2, \ldots$ in Section 2.4 this means that there are too many points close to the origin at the beginning of the sequence.

This "leading-zeros phenomenon" can be alleviated by throwing away a certain number of initial terms of the sequence $q_1, q_2, \ldots$. Clearly, the number thus skipped should be some power of b. The argument in the previous paragraph indicates that this power should be at least the maximum value of e, which grows like $\log s$ as noted earlier. Even allowing some "fat" for the implicit constant, this is feasible. We warm up this way taking a time which for practical purposes can be considered to be constant, jumping over the skipped values as Section 4.2 sketches.

We may further alleviate the leading-zeros phenomenon by changing the initial values $v_0, \ldots, v_{m-1}$ of the linear recursion of order $m = e(q + 1)$. We introduce the following general method of defining these initial values. Choose a sequence $K_0, K_1, \ldots$ of integers with $0 \le K_q < e(q + 1)$ for all $q \ge 0$ and $\lim_{q \to \infty} K_q = \infty$. For any $q \ge 0$ choose initial values $v(q, r)$, $r = 0, 1, \ldots, e(q + 1) - 1$, as follows. First consider the case where $K_q < eq$. Then put $v(q, r) = 0$ for $0 \le r < K_q$ and $v(q, K_q) = 1$, and let $v(q, r)$ be arbitrary for $K_q < r < eq$. Let

$$p(x)^q = x^{eq} - h_{eq-1}x^{eq-1} - \cdots - h_0.$$

Then let

$$v(q, eq) \neq \operatorname*{ADD}_{r=0}^{eq-1} \mathrm{MUL}(h_r, v(q, r))$$

be otherwise arbitrary and let $v(q, r)$ be arbitrary for $eq < r < e(q + 1)$. Now consider the case where $K_q \ge eq$. Here we choose $v(q, r) = 0$ for $0 \le r < K_q$ and $v(q, K_q) = 1$, and let $v(q, r)$ be arbitrary for $K_q < r < e(q + 1)$. Making these choices brings the algorithm in Section 2.3 into final form.

The conditions of [13, Theorem 1] are satisfied, and so the resulting sequence $q_1, q_2, \ldots$ is a (t, s)-sequence in base b with the same value of t as the sequence with the original choice of initial values for the v_i. The original choice of initial values in [13] corresponds to the maximal value $K_q = e(q + 1) - 1$ in the present setup. By choosing smaller values for K_q, we alleviate the leading-zeros phenomenon. Our program currently sets each K_q equal to eq. This has the effect of setting all unrestricted values of v to 1.

3.4 Practical Restrictions on s and b

For large b, the initial segments of the sequences are very nonuniform, except when the length of the segment is close to a power of b. When b is the smallest prime $\geq s$, the Niederreiter and Faure sequences coincide. Fox [6] informally discusses such nonuniformity in this case. Restricting attention to base 2 does not get around the problem, because $B(s, 2)$ becomes large.

A more fundamental problem is that the factor $\Delta(N)/N$ in the Koksma-Hlawka error bound for quasi-Monte Carlo integration [11] becomes huge for $s > 12$, say, and practical values of N. Thus, the only possible stopping rules are heuristic: (a more precise version of) stop when successive estimates show little variation. That can be dangerous.

An alternative uses pseudo-Monte Carlo, where the central limit theorem implies that errors of order larger than $N^{-1/2}$ are very unlikely. For large s, this meaningful probabilistic statement is better than a useless deterministic bound.

For these reasons, our current program restricts s to at most 12. Changing one parameter throughout the suite of programs and adding appropriate tables (in principle, easy) allows essentially arbitrary s. A similar remark applies to b. Since we do not recommend using larger s and b than we currently allow, we omit details about extending the parameter ranges in the programs.

The point of allowing larger bases than would seem theoretically worthwhile (for $s < 13$) is to allow experimentation. Recall that the theory minimizes error *bounds*. It is not hard to allow for large bases, say up to 50.

4. REMARKS ON THE PROGRAMS

4.1 The Two Implementations

We use two different programs to implement Niederreiter's sequence generator, one for sequences in any prime-power base, the other only for sequences in base 2. The programs share a common structure. Before generating a sequence, the user must call an initialization routine which in turn calls a routine to calculate the values of the elements c. Thereafter, each call on the generating routine returns a new quasirandom vector. The general program does field arithmetic using table look-up, reading precomputed addition and multiplication tables and a table of irreducible polynomials from a file; the base-2 program does arithmetic modulo 2 and the necessary irreducible polynomials are supplied in DATA statements.

The base-2 generator is an order-of-magnitude faster than the general case. This speed is obtained at the price of using a bit-by-bit exclusive-or operation not provided by standard FORTRAN [1]. However, most FORTRAN compilers include an appropriate operator or implicit function. Our base-2 program uses an implicit function called in just three places. It could be replaced by a function in standard FORTRAN, but the resulting drastic loss of speed would remove any advantage the base-2 program has over the general program.

4.2 The General Program

This program, entirely in standard FORTRAN, consists of four principal subroutines, *INLO*, *CALCC*, *CALCV*, and *GOLO*, and five auxiliary subroutines.

The subroutine *INLO* should be called once for each sequence of points to be generated. (The program as written can generate several different sequences of points in the same run, but not concurrently.) The parameter *DIMEN* gives the dimension of the quasirandom vectors to be generated. Unless $1 \leq DIMEN \leq 12$, the subroutine stops with an error message. The parameter *BASE* gives the order of the field to be used. The parameter *SKIP* says how many values at the beginning of the sequence are to be omitted; this is done implicitly so the work has order log(*SKIP*).

INLO calls *SETFLD*, which initializes addition, multiplication, and subtraction tables for the field of order $Q = BASE$. The tables are calculated if Q is a prime, or read from a file if Q is a composite prime power. (This file must be calculated beforehand; the program *GFARIT* will do this: see Appendix A.) *INLO* next decides how many base-Q digits can conveniently be represented (this is R in Section 2.2) and calls *CALCC* to compute the elements c (see Section 2.3). It further initializes arrays *COUNT* (corresponding to a in our description of the generator), D (corresponding to $d_j^{(i)}$), and *NEXTQ* (corresponding to $Q_n^{(i)}$) as well as *RECIP* (b^{-R}) and other auxiliary variables.

The subroutine *CALCC* calculates the elements $c_{jr}^{(i)}$ as outlines in Section 2.3. The array C of values to be returned to the calling program, and the number of dimensions and of base-Q digits in use, are passed through a common block. The algorithm of Section 2.3, as modified in Section 3.3, is programmed straightforwardly. Irreducible polynomials $p^{(i)}(x)$ over F_Q are read from a file (which must be calculated beforehand; the program *GFPLYS* will do this: see Appendix A). The subroutine *CALCV* computes powers of $p^{(i)}(x)$ and calculates the required values v_l.

After the initial call on *INLO*, quasirandom vectors are generated by calling *GOLO*. Each call returns a new quasirandom vector. Necessary information (the dimension of the vector, the base in use, etc.) is saved in a common block at initialization and between calls.

GOLO calculates the new quasirandom vector using the values saved in *NEXTQ* and updates the latter as follows. First, array *COUNT* that gives the base-Q representation of $n - 1$ is updated by adding 1 to the units digit and propagating the carry. Array D is holding the previous values of $d_j^{(i)}$. For each digit of *COUNT* that changes, a new value of D is computed. Usually the carry propagates only a few places along *COUNT*, so this does not take long. (In general, the greater the value of Q, the less carries there are.) Then we use the new values of D to calculate *NEXTQ* ready for the next call on *GOLO*.

4.3 The Base-2 Program

Like the general program, the base-2 version consists of four principal subroutines, *INLO*2, *CALCC*2, *CALCV*, and *GOLO*2. (The suffix 2 labels

subroutines tailored to base 2.) It uses a nonstandard function to perform bit-by-bit exclusive-or operations on integer variables. This is called twice in *INLO2* and once in *GOLO2*. Some subroutines, such as *CALCV*, are the same for base 2 and general base. Unlike the general program, the self-contained base-2 version does not read precomputed tables from a file.

The base-2 program differs from the general program in two other major respects. First, since the elements of F_2 are bits, we pack multiple bits into one computer word for greatly increased efficiency via the bitwise exclusive-or. Second, taking the Gray-code representation of $n - 1$ instead of the ordinary binary representation further improves the generator.

INLO2 is similar to *INLO*. It is not necessary here to specify the base in use. Arithmetic in F_2 is simply arithmetic modulo 2, so precalculated tables are not needed. The number of base-2 digits in use, here called *NBITS*, is defined as the number of bits in an integer variable (not counting the sign). *INLO2* uses *CALCC2* to compute the elements $c_{jr}^{(i)}$, clears *COUNT* (here an integer variable, not an array) and *NEXTQ*, and exits. No array D is needed.

Like its counterpart, *CALCC2* calculates the elements $c_{jr}^{(i)}$ as outlined in Section 2.3, using arithmetic modulo 2. The coefficients of the required irreducible polynomials $p^{(i)}(x)$ over F_2 are given in a DATA statement. The same auxiliary subroutine *CALCV* as before computes powers of $p^{(i)}(x)$ and the required values v_i. Here, however, *CALCC2* initializes not a three-dimensional array C but a two-dimensional one CJ. As elements $c_{jr}^{(i)}$ are computed, they are packed into this array so that $CJ(I, R)$ holds the values of $c_{jr}^{(i)}$ for j varying from 1 to *NBITS* (which corresponds to R in Section 3.1). This is the format required to exploit the Gray-code technique discussed in Section 3.1.

After an initial call on *INLO2* quasirandom vectors are generated by calling *GOLO2*. As in the general case, each call returns a new quasirandom vector. *GOLO2* calculates the new vector using the values saved in *NEXTQ*, then locates the rightmost zero-bit in *COUNT* (corresponding to $n - 1$). Now each word in the array *NEXTQ* is brought up-to-date by a single exclusive-or operation with the corresponding word of *CJ* (see Section 3.1).

Remarkably, the subroutines for generating a new quasirandom vector using Niederreiter's technique in base 2 and using Sobol's method are essentially identical. In both cases the new vector is generated using an array of numerators, here called *NEXTQ*, and then this array is updated by a single exclusive-or operation on each of its members. Only the constants used in this updating differ, depending on whether they were calculated by the initialization routine for Niederreiter's method or for Sobol's. It would even be possible to calculate and store both sets of constants, and then to use the same subroutine to generate quasirandom vectors by either method. More generally, *GOLO2* can operate with any constants (called, generically, *direction numbers* by Sobol') replacing Sobol's or Niederreiter's constants. Since *GOLO2* is so fast, it may pay to introduce such constants artificially into methods where they do not appear naturally–such as that of Hansen et al. [8].

Of course, Sobol' offers no counterpart to Niederreiter's technique for bases other than $b = 2$.

5. TEST RESULTS

5.1 Test Integrals

We tested our programs on the four multidimensional integrals below:

$$I_1 = \int_0^1 \cdots \int_0^1 \prod_{i=1}^{s} |4x_i - 2|\, dx_1 \cdots dx_s = 1,$$

$$I_2 = \int_0^1 \cdots \int_0^1 \prod_{i=1}^{s} i \cos(ix_i)\, dx_1 \cdots dx_s = \prod_{i=1}^{s} \sin i,$$

$$I_3 = \int_0^1 \cdots \int_0^1 \prod_{i=1}^{s} T_{n_i}(2x_i - 1)\, dx_1 \cdots dx_s = 0,$$

where T_n is the Tchebyshev polynomial of degree n and $n_i = (i \bmod 4) + 1$, and

$$I_4 = \int_0^1 \cdots \int_0^1 \sum_{i=1}^{s} (-1)^i \prod_{j=1}^{i} x_j\, dx_1 \cdots dx_s = -\frac{1}{3}\left(1 - \left(-\frac{1}{2}\right)^s\right).$$

For the integrals I_1, I_3, and I_4, the answers we got are reasonably close to the respective exact values, giving us confidence that our programs implement the sequences correctly. For I_2 with $s = 8$, the answers we got are wildly erratic. However, for I_2 and $s = 2$ or 3, our answers seem to be converging to the exact value. Thus, the erratic behavior in higher dimensions is, we believe, due to the intrinsic volatility of the integral and not to a programming error. Beyond this, the answers indicate how well the sequences perform on these integrals—at least for the combinations of dimension, base, sequence length, and skip values tried.

Tables I–IV display these results. The odd-looking iteration numbers are powers of the indicated base. The set of tables lists results for the integrals I_1, I_2, I_3, and I_4 in that order with base-2 and "optimal"-base output side-by-side; as discussed earlier and indicated in the tables, the word "optimal" here appears to be a misnomer.

All base-2 results were obtained with the tailored programs. Except at powers of two, the results with base 2 using the general program would be slightly different because Gray code shuffles the sequence.

For the test integral 4, the Sobol' sequence produced the estimate -0.3320198 using 50000 points. In our experiments, this is typical of the closeness of the results of the Sobol' sequence and the Niederreiter base-2 sequence (whenever both are reasonably accurate). We also tried $b = 11$ with $SKIP = 14641$; the results are roughly as accurate as those for $b = 9$.

Section 5.2 briefly discusses these tables, with conclusions deferred to Section 6.

5.2 Accuracy and Speed

The tables in Section 5.1 indicate that base 2 is at least as good as the "optimal" base in terms of accuracy; sometimes it is much better. Taking

Table I
Test integral 1, DIMEN = 8
Correct value is 1.000000

BASE = 2 SKIP = 4096		BASE = 9 SKIP = 6561	
Iteration	Estimate of integral	Iteration	Estimate of integral
10000	1.004320	10000	0.9495928
16384	0.9996442	20000	0.9926008
20000	0.9978413	50000	1.020338
32768	0.9933131	59049	1.017750
50000	0.9959128		

Table II
Test integral 2, DIMEN = 8
Correct value is −0.1423184E − 01

BASE = 2 SKIP = 4096		BASE = 9 SKIP = 6561	
Iteration	Estimate of integral	Iteration	Estimate of integral
10000	19.95929	10000	−32.56793
16384	19.96111	50000	−42.56069
32768	17.11567	59049	−26.86514
50000	10.42192		

iteration numbers along powers of the base has little effect for base 2, but can have a significant helpful effect for the "optimal" base. Some integrals are just too hard to evaluate with reasonable accuracy with a reasonable number of points. This underlines the importance of looking at the results at a sequence of iteration numbers, to judge (heuristically) whether the results appear to be converging; but recall our caveat in Section 3.4.

In each case, the results from the tailored base-2 programs were obtained at least several times faster. For easily-evaluated integrands, this is another reason to prefer base 2. For hard-to-evaluate integrands, the speed of producing quasirandom points becomes irrelevant but—since the accuracy with

Table III
Test integral 3, DIMEN = 8
Correct value is 0.0000000E + 00

BASE = 2 SKIP = 4096		BASE = 9 SKIP = 6561	
Iteration	Estimate of integral	Iteration	Estimate of integral
10000	−0.2915053E − 04	10000	−0.2077495E − 04
16384	−0.1257072E − 04	20000	−0.3812500E − 04
20000	−0.8120412E − 04	50000	−0.1236383E − 03
32768	−0.2067957E − 04	59049	−0.1133050E − 03
50000	−0.4909454E − 04		

Table IV
Test integral 4, DIMEN = 8
Correct value is −0.3320313

BASE = 2 SKIP = 4096		BASE = 9 SKIP = 6561	
Iteration	Estimate of integral	Iteration	Estimate of integral
10000	−0.3320505	10000	−0.3315212
16384	−0.3320175	20000	−0.3318718
20000	−0.3320289	50000	−0.3317828
32768	−0.3320215	59049	−0.3318894
50000	−0.3320297		

base 2 is at least as good as for other bases (if the results above can be safely extrapolated)—base 2 remains the best choice.

The speeds for the Sobol' programs and the base-2 programs implementing Niederreiter's sequences are exactly the same. For the integrals above, they produce results roughly comparable in terms of accuracy.

6. CONCLUSIONS

We speculate that the results in Section 5 are broadly representative of what would happen with far more extensive experimentation. Assuming that this

is so, among Niederreiter's sequences, we recommend base 2. Since our programs tailored for base 2 are much faster and involve only a slight loss of portability, we recommend them; the adjustments required as they are moved from one computer to another take at most a few minutes to make.

Between the base-2 Niederreiter sequence and the Sobol' sequence, the empirical results in Section 5 offer essentially no guidance. As indicated earlier in this paper, however, the former is strictly better theoretically when the dimension exceeds seven and is comparable theoretically otherwise. Therefore, we would choose the base-2 Niederreiter sequence when the dimension exceeds seven. To avoid having multiple sets of programs, we would choose that sequence for all dimensions.

Even among integrands of bounded variation, some are just too volatile to handle by any method. In high-dimensional problems (say > 12), quasi-Monte Carlo seems to offer no practical advantage over pseudo-Monte Carlo because the discrepancy bound for the former is far larger than $N^{-1/2}$ for $N = 2^{30}$, say. Since the bound can be loose, quasi-Monte Carlo can be tried as a heuristic in high dimensions—but with no practical idea of the order of magnitude of the error, in contrast to pseudo-Monte Carlo. We think that it is better to have a realistic probabilistic estimate (available from the central limit theorem with pseudo-Monte Carlo) rather than to have no realistic estimate at all.

APPENDIX A

Field-arithmetic tables for use by the generator are created by the programs *GFARIT* and *GFPLYS* and their respective subroutines. *GFARIT* calculates addition, multiplication, and subtraction tables for fields whose order is p^k for some prime p and $k > 1$, up to an arbitrary order currently fixed at 50; *GFPLYS* calculates irreducible polynomials for all the finite fields of order less than this same arbitrary order. The results are written out to files, called gftabs.dat and irrtabs.dat in our implementation. After gftabs.dat and irrtabs.dat are produced, *GFARIT* and *GFPLYS* are no longer needed. Since *GFPLYS* reads gftabs.dat, *GFARIT* must be run first. Subroutines of the driver *GENIN* read both gftabs.dat and irrtabs.dat to produce quasirandom vectors.

To calculate the field-arithmetic tables for a field of order $q = p^k$, the integers $0, 1, \ldots, q - 1$ are taken to correspond in the obvious way to polynomials with coefficients in F_p: the integer

$$i = a_{k-1}p^{k-1} + a_{k-2}p^{k-2} + \cdots + a_0$$

corresponds to the polynomial

$$a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_0$$

and vice versa. On the machine, polynomials are stored as an array of coefficients. Three subroutines *PLYADD*, *PLYMUL*, and *PLYDIV* add, multiply, and divide polynomials in this form; two others, *ITOP* and *PTOI*, convert back and forth between integers and the corresponding polynomials.

For each value of $q = p^k$ of interest (currently 4, 8, 9, 16, 25, 27, 32, and 49) an irreducible polynomial of degree k with coefficients in F_p is provided in a DATA statement. Calculating field-arithmetic tables is now straightforward: to find the sum of elements i and j of the field of order q, convert them both to polynomials, add, and convert back to an integer; to multiply these elements, convert them to polynomials, multiply, divide the result by the appropriate irreducible polynomial, and convert the remainder back to an integer. The subtraction table is obtained from the addition table (and, were it necessary, a division table could be obtained from the multiplication table).

To calculate irreducible polynomials over F_q, the field of order q, *GFPLYS* uses a sieve. Here integers correspond to polynomials with coefficients in F_q. An array is initialized to hold integers of the form

$$q^k + a_{k-1}q^{k-1} + \cdots + a_0, \quad k = 1, 2, \ldots,$$

that is, integers which correspond to monic polynomials, in order of increasing degrees. Now a simple adaptation of the sieve of Eratosthenes [14, Chap. 1] allows us to pick out those which correspond to irreducible polynomials. To delete "multiples" of an integer, we convert it to a polynomial, calculate the multiples of the result, and convert these back to integers, using the same routines *ITOP*, *PLYMUL*, and *PTOI* mentioned above. Sieving continues until a sufficient number of irreducible polynomials has been found. Among those of a fixed degree, the order may differ from those in published tables. This is important when comparing results produced by our general-base program for base 2 with results produced by our base-2 programs, which use published tables. To implement the sieve in subroutine *IRRED*, once an irreducible polynomial is found, its multiples are deleted. The integer function *FIND* searches an array of polynomials (coded as integers) for a particular item. If it succeeds, the item is removed from the array; otherwise, no action is needed.

APPENDIX B

A small numerical example may help to illustrate the ideas outlined in Section 2.3. Suppose then that we are working in base $b = 3$, that our machine can represent integers less than 3^4 in single precision, so $R = 4$, and that our chosen polynomial $p(x)$ is $x^2 + 1$, so $e = 2$.

Throughout the example, all arithmetic is to be understood as taking place in F_3 unless we indicate the contrary; in other words, addition, subtraction, and multiplication are all taken modulo 3. For notational convenience, we use $=$ rather than the more correct $\equiv$, even going so far as to write such things as $2 = -1$.

The algorithm for calculating the elements c proceeds as follows:

(1) The first time through step (3) $j = 1$, and we begin by setting $q \leftarrow 0$, $u \leftarrow 0$. We calculate

$$b(x) = p(x)^1 = x^2 + 1 = x^2 - 2,$$

so $m = 2$, $b_0 = 2$, and $b_1 = 0$. Hence $v_0 = 0$, $v_1 = 1$, and from $v_{r+2} = 2v_r, r = 0, 1, \ldots$, we find $v_2 = v_4 = 0$ and $v_3 = 2$. Now $c_{1r} = v_r$, and so

$$c_{10} = 0, \quad c_{11} = 1, \quad c_{12} = 0, \quad c_{13} = 2.$$

(2) The second time through the loop, $u = 1$ and step (3) is omitted. In step (4) we therefore use the same values of v to find $c_{2r} = v_{r+1}$, i.e.,

$$c_{20} = 1, \quad c_{21} = 0, \quad c_{22} = 2, \quad c_{23} = 0.$$

(3) The third time through the loop we arrive at step (3) with $j = 3$. We set $q \leftarrow 1$ and $u \leftarrow 0$, and calculate

$$b(x) = p(x)^2 = x^4 + 2x^2 + 1 = x^4 - x^2 - 2.$$

Now $m = 4$, $b_0 = 2$, $b_1 = b_3 = 0$, and $b_2 = 1$, so $v_0 = v_1 = v_2 = 0$, $v_3 = 1$, and from $v_{r+4} = v_{r+2} + 2v_r$, $r = 0, 1, \ldots$, we find $v_4 = 0$. Now $c_{3r} = v_r$, and so

$$c_{30} = 0, \quad c_{31} = 0, \quad c_{32} = 0, \quad c_{33} = 1.$$

(4) On the final pass through the loop the values of v remain unchanged, and we have $c_{4r} = v_{r+1}$, giving

$$c_{40} = 0, \quad c_{41} = 0, \quad c_{42} = 1, \quad c_{43} = 0.$$

Suppose we want to use these values of c to generate x_8, for example. We have $n - 1 = 7$ (in decimal) = 0021 (in base 3), so $a_0 = 1$, $a_1 = 2$, and $a_2 = a_3 = 0$. Thus

$$d_1 = c_{10}a_0 + c_{11}a_1 + c_{12}a_2 + c_{13}a_3 = 2,$$

and similarly $d_2 = 1$, $d_3 = 0$, and $d_4 = 0$. Hence

$$Q_8 = 2100 \quad \text{(in base 3)} = 63 \quad \text{(in decimal)}$$

and $x_8 = Q_8/3^4 = 63/81$ (in decimal) = 0.77... (decimal).

NOTE

The programs are available via electronic mail from the authors. The package includes sample drivers (*GENIN* and *GENIN2*), a directory of programs, and a guide to the programs.

ACKNOWLEDGMENTS

We thank Russell Cheng and Teresa Davenport whose suggestions and advice helped to improve our programs. Part of this work was carried out while Paul Bratley was visiting the Max-Planck-Institut für Geschichte, Göttingen, and while Bennett Fox was visiting the Operations Research Department of AT & T Bell Laboratories. In each case, the hospitality of the host institution is greatly appreciated. In particular, Bratley thanks Manfred Thaller and Fox thanks Paul Glasserman.

One version of the paper was written while all three authors were at the conference on Random Number Generation and Quasi-Monte Carlo Methods held at the University of Alaska, Fairbanks. We particularly thank Pat Lambert for making this possible.

REFERENCES

1. AMERICAN NATIONAL STANDARDS INSTITUTE, INC. *American National Standard Programming Language FORTRAN*. Standard X3.9-1978, New York, 1978.
2. ANTONOV, I. A., AND SALEEV, V. M. An economic method of computing LP_τ-sequences. *USSR Comput. Math. Math. Phys. 19*, 1 (1979), 252–256.
3. BRATLEY, P., AND FOX, B. L. Algorithm 659: Implementing Sobol's quasi-random sequence generator. *ACM Trans. Math. Softw. 14* (1988), 88–100.
4. DAVENPORT, T. R., AND CHENG, R. C. H. Implementation of quasi-random generators and their use in discrete event simulation. In *Proceedings of the 1989 Winter Simulation Conference* (Washington, DC, Dec. 4–6, 1989), E. A. MacNair, K. J. Musselman, and P. Heidelberger, Eds., pp. 419–427.
5. FAURE, H. Discrépance de suites associées à un système de numération (en dimension s). *Acta Arith. 41* (1982), 337–351.
6. FOX, B. L. Algorithm 647: Implementation and relative efficiency of quasirandom sequence generators. *ACM Trans. Math. Softw. 12* (1986), 362–376.
7. HALTON, J. H. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numer. Math. 2* (1960), 84–90; and Correction *ibid 2* (1960), 196.
8. HANSEN, T., MULLEN, G. L., AND NIEDERREITER, H. Good parameters for a class of node sets in quasi-Monte Carlo integration. To appear, *Math. Comput.*
9. HUA, L. K., AND WANG, Y. *Applications of Number Theory to Numerical Analysis*. Springer, Berlin, 1981.
10. LIDL, R., AND NIEDERREITER, H. *Finite Fields*. Addison-Wesley, Reading, Mass., 1983. (Now distributed by Cambridge University Press, Cambridge, 1984.)
11. NIEDERREITER, H Quasi-Monte Carlo methods and pseudo-random numbers. *Bull. Am. Math. Soc. 84* (1978), 957–1041.
12. NIEDERREITER, H. Point sets and sequences with small discrepancy. *Monatsh. Math. 104* (1987), 273–337.
13. NIEDERREITER, H. Low-discrepancy and low-dispersion sequences. *J. Number Theor. 30* (1988), 51–70.
14. RIESEL, H. *Prime Numbers and Computer Methods for Factorization*. Birkhäuser, Boston, 1985.
15. SOBOL', I. M. The distribution of points in a cube and the approximate evaluation of integrals. *USSR Comput. Math. Math. Phys. 7*, 4 (1967), 86–112.

Received March 1992; revised September 1992; accepted October 1992