ALGORITHM 659
Implementing Sobol's Quasirandom Sequence Generator

PAUL BRATLEY and BENNETT L. FOX
Université de Montréal

We compare empirically accuracy and speed of low-discrepancy sequence generators of Sobol' and Faure. These generators are useful for multidimensional integration and global optimization. We discuss our implementation of the Sobol' generator.

Categories and Subject Descriptors: G.1.4 [**Numerical Analysis**]: Quadrature and Numerical Differentiation—*multidimensional integration*; G.4 [**Mathematics of Computing**]: Mathematical Software—*certification and testing, efficiency*

General Terms: Algorithms, Measurement, Performance

Additional Key Words and Phrases: Computational complexity, discrepancy, generators, global optimization, multidimensional integration, quasirandom sequences

1. INTRODUCTION

We want to generate a set of points $x^1, x^2, \ldots, x^N$ well spaced in the s-dimensional unit cube $I^s = [0, 1]^s$. One measure of good spacing is discrepancy, defined as follows:

Given a set of points $x^1, x^2, \ldots, x^N \in I^s$ and a subset $G \subset I^s$, define the counting function $S_N(G)$ as the number of points $x^i \in G$. For each $x = (x_1, x_2, \ldots, x_s) \in I^s$, let G_x be the rectangular s-dimensional region

$$G_x = [0, x_1) \times [0, x_2) \times \cdots \times [0, x_s)$$

with volume $x_1 x_2 \cdots x_s$. Then the discrepancy of the points $x^1, x^2, \ldots, x^N$ is

$$D^*(x^1, x^2, \ldots, x^N) = \sup_{x \in I^s} | S_N(G_x) - N x_1 x_2 \cdots x_s |.$$

The smaller the discrepancy, the better the spacing.

It is believed, though it has not been proved, that no s-dimensional infinite sequence can have discrepancy an order of magnitude smaller than $\log^s N$. Sequences that aim for low discrepancy are (misleadingly) called *quasirandom*.

This work was supported by the Natural Sciences and Engineering Research Council of Canada.
Authors' address: Département d'informatique et de recherche opérationnelle, Université de Montréal, P.O. Box 6128, Station A, Montreal, Canada H3C 3J7.

They are used in numerical integration, simulation, and optimization [13]. Several ways of generating such sequences are known: Fox [4] compares efficiency and ease of implementation for sequences suggested by Halton [5] and Faure [3]. The comparison is extended to include a standard pseudorandom sequence generator, and a third quasirandom sequence generator suggested by Sobol' [10] is mentioned briefly.

The present paper describes an implementation of Sobol's generator, and compares it in some detail to the generator suggested by Faure, which Fox [4] finds preferable to Halton's technique. For an account of the way the Faure generator was implemented, see Fox [4]. Sarkar and Prasad [9] apparently initialize the Faure sequence following [3]. As described and justified on pages 369–370 of [4], Fox uses a much different (advanced) start. This probably accounts for their (different) conclusion in [9] that Halton and Faure sequences "perform equally well." They do not consider the Sobol' sequence.

The Halton, Faure, and Sobol' sequences all have $O(\log^s N)$ discrepancies. Implicit proportionality factors, depending on s, differ. For fixed $s \geq 5$ and large enough N, these factors from smallest to largest correspond to the Faure, Sobol', and Halton sequences, respectively [3]. Recently, Niederreiter [8] proposed a sequence with smallest such proportionality factor currently known. By itself, the asymptotic ranking of these factors does not tell the whole story as Fox [4], for example, discusses.

2. THE SOBOL' GENERATOR

The theoretical underpinnings of Sobol's method of generating quasirandom sequences can be found in [10]. Here we give only an informal description of the necessary computations.

To begin, suppose we are working in just one dimension. Our aim is thus to generate a sequence of values $x^1, x^2, \ldots, 0 < x^i < 1$, with low discrepancy over the unit interval. We first need a set of *direction numbers* $v_1, v_2, \ldots$. Each v_i is a binary fraction that can be written in either of two ways:

$$v_i = 0.v_{i1}v_{i2}v_{i3}\cdots$$

where v_{ij} is the jth bit following the binary point in the expansion of v_i; or alternatively,

$$v_i = m_i/2^i$$

where m_i is an odd integer, $0 < m_i < 2^i$. In particular, we see from the second representation that $v_{ij} = 0$ if $j > i$.

Table I provides a simple illustration of these ideas.

So where do the v_i come from? To obtain them, we begin by choosing a polynomial with coefficients chosen from {0, 1}, which is a primitive polynomial [6] in the field Z_2. Thus we might choose, say,

$$P \equiv x^d + a_1x^{d-1} + \cdots + a_{d-1}x + 1$$

where each a_i is 0 or 1 and P is a primitive polynomial of degree d in Z_2. (If P is primitive, then a_d, the constant term, is necessarily equal to 1.) Provided P is primitive, the choice of polynomial is otherwise arbitrary. The number of primi-

Table I. Direction Numbers

i	1	2	3	4	5	6
m_i	1	3	7	5	7	43
v_i (in binary)	0.1	0.11	0.111	0.0101	0.00111	0.101011

tive polynomials of degree d is $\phi(2^d - 1)/d$, where ϕ is the Euler function, and they are widely tabulated (e.g., see [7]).

Once we have chosen a polynomial, we use its coefficients to define a recurrence for calculating v_i; thus,

$$v_i = a_1 v_{i-1} \oplus a_2 v_{i-2} \oplus \cdots \oplus a_{d-1} v_{i-d+1} \oplus v_{i-d} \oplus [v_{i-d}/2^d], \qquad i > d$$

where $\oplus$ denotes a bit-by-bit exclusive-or operation, and the last term is v_{i-d} shifted right d places. Equivalently, we may express the recurrence in terms of the m_i and calculate

$$m_i = 2a_1 m_{i-1} \oplus 2^2 a_2 m_{i-2} \oplus \cdots \oplus 2^{d-1} a_{d-1} m_{i-d+1} \oplus 2^d m_{i-d} \oplus m_{i-d}.$$

This is the form of the recurrence used in our implementation, since it is easily programmed using integer arithmetic.

Using a primitive polynomial of degree d, the values of $m_1, m_2, \ldots, m_d$ can be chosen freely provided that each m_i is odd and $m_i < 2^i$; subsequent values m_{d+1}, $m_{d+2}, \ldots$ are then determined by the recurrence. For example, the values in Table I were calculated using the primitive polynomial

$$x^3 + x + 1$$

of degree 3. The corresponding recurrence is

$$m_i = 4m_{i-2} \oplus 8m_{i-3} \oplus m_{i-3},$$

and the initial values $m_1 = 1$, $m_2 = 3$, and $m_3 = 7$ were chosen for reasons set out below. Then

$$\begin{aligned} m_4 &= 12 \oplus 8 \oplus 1 \\ &= 1100 \oplus 1000 \oplus 0001 \qquad \text{in binary} \\ &= 0101 \quad \text{in binary} \\ &= 5 \\ m_5 &= 28 \oplus 24 \oplus 3 \\ &= 11100 \oplus 11000 \oplus 00011 \qquad \text{in binary} \\ &= 00111 \quad \text{in binary} \\ &= 7 \\ m_6 &= 20 \oplus 56 \oplus 7 \\ &= 010100 \oplus 111000 \oplus 000111 \qquad \text{in binary} \\ &= 101011 \quad \text{in binary} \\ &= 43 \end{aligned}$$

and so on.

Finally, to generate the sequence $x^1, x^2, \ldots$ we can use

$$x^n = b_1 v_1 \oplus b_2 v_2 + \cdots$$

where $\ldots b_3b_2b_1$ is the binary representation of n. This is Sobol's original method. Antonov and Saleev [1] prove that taking

$$x^n = g_1v_1 \oplus g_2v_2 \oplus \cdots \qquad (*)$$

where $\ldots g_3g_2g_1$ is the Gray code representation of n does not affect the asymptotic discrepancy. For $k = 2, 3, \ldots$, it shuffles each initial segment of length 2^k of Sobol's original sequence. We remind the reader of the following properties of the Gray code:

(i) The Gray code for n is obtained from the binary representation of n using

$$\ldots g_3g_2g_1 = \ldots b_3b_2b_1 \oplus \ldots b_4b_3b_2.$$

(ii) The Gray code for n and the Gray code for $n + 1$ differ in only one position. If b_c is the rightmost zero-bit in the binary representation of n (add a leading zero to n if there are no others), then g_c is the bit whose value changes.

Using these properties, and defining x^n by (*), we can calculate x^{n+1} in terms of x^n as

$$x^{n+1} = x^n \oplus v_c \qquad (**)$$

where b_c is the rightmost zero-bit in the binary representation of n. The Antonov–Saleev method is thus much faster than Sobol's original scheme. To start the recurrence, we take $x^0 = 0$.

Returning to the example of Table I, the first few values of x are thus generated as follows:

Initialization:	$x^0 = 0$	
	$n = 0$	in binary so
	$c = 1$	
Step 1:	$x^1 = x^0 \oplus v_1$	
	$= 0.0 \oplus 0.1$	in binary
	$= 0.1$	in binary
	$= \frac{1}{2}$	
	$n = 1$	in binary so
	$c = 2$	
Step 2:	$x^2 = x^1 \oplus v_2$	
	$= 0.10 \oplus 0.11$	in binary
	$= 0.01$	in binary
	$= \frac{1}{4}$	
	$n = 10$	in binary so
	$c = 1$	
Step 3:	$x^3 = x^2 \oplus v_1$	
	$= 0.01 \oplus 0.10$	in binary
	$= 0.11$	in binary
	$= \frac{3}{4}$	
	$n = 11$	in binary so
	$c = 3$	

and so on. Of course the values of x^n may also be obtained directly from the definition (*). Thus for $n = 23$, for example, we find

$$\begin{aligned} n &= 10111 && \text{in binary} \\ \mathit{Graycode}(n) &= 10111 \oplus 01011 \\ &= 11100 \\ \text{and hence } x^{23} &= v_3 \oplus v_4 \oplus v_5 \\ &= 0.11100 \oplus 0.01010 \oplus 0.00111 \\ &= 0.10001 && \text{in binary} \\ &= \tfrac{17}{32} \end{aligned}$$

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

$$q^n = (x_1^n, x_2^n, \ldots, x_s^n).$$

Sobol' [10] proves that, to get $O(\log^s N)$ discrepancy, it suffices to choose any s different primitive polynomials, to calculate s different sets of direction numbers as explained above, and then to generate each component x_i^n of the quasirandom vector separately using the corresponding set of direction numbers.

3. REMARKS ON THE SOBOL' GENERATOR

3.1 A Special Case

As well as the direction numbers obtained from primitive polynomials, Sobol' [10] shows that a suitable one-dimensional sequence x^n can be obtained by taking $m_i = 1$ for all i. This sequence may be used to generate one component of a quasirandom vector as outlined above.

3.2 Choice of Initial Values

When using a primitive polynomial of degree d to calculate a set of direction numbers, the d values m_i, $1 \le i \le d$, can be chosen freely provided that each m_i is odd and that $m_i < 2^i$, $1 \le i \le d$. How should we choose these values?

It is clearly desirable that the m_i used to generate different components of a quasirandom vector q^n should themselves differ. Otherwise, since the initial values $m_1, m_2, \ldots$ determine the initial values of the sequence x^n generated, several components of the quasirandom vector behave identically for small values of n. Although asymptotically this may not matter, it is of practical concern if only short sequences are to be used. (If two components x_i^n and x_j^n of q^n use the same initial values $m_1, m_2, \ldots, m_k$, then $x_i^n = x_j^n$ for $1 \le n \le 2^k - 1$.)

Sobol' [11] gives additional uniformity properties for quasirandom vectors which can be obtained by suitable choice of the m_i. Changing notation for a moment, let v_{ijk} denote the kth bit after the binary point of the jth direction number used to generate the ith component of an s-dimensional quasirandom vector. Sobol's "property A" is obtained if

$$\det \begin{vmatrix} v_{111} & v_{121} & \cdots & v_{1s1} \\ \vdots & \vdots & & \vdots \\ v_{s11} & v_{s21} & \cdots & v_{ss1} \end{vmatrix} = 1.$$

His property A′ is obtained if

$$\det \begin{vmatrix} v_{111} & v_{121} & \cdots & v_{1,2s,1} \\ v_{112} & v_{122} & \cdots & v_{1,2s,2} \\ \vdots & \vdots & & \vdots \\ v_{s11} & v_{s21} & \cdots & v_{s,2s,1} \\ v_{s12} & v_{s22} & \cdots & v_{s,2s,2} \end{vmatrix} = 1.$$

(All arithmetic is to be carried out modulo 2.) Property A holds for subsequences of length 2^s, property A′ for subsequences of length 2^{2s}; they are therefore of less practical interest as s increases. Sobol' and Levitan [14] give m_i that satisfy the conditions for property A when $s \leq 16$ and the conditions for property A′ when $s \leq 6$ (but see Section 4.4).

3.3 Computational Complexity

We ignore the work required to calculate necessary direction numbers during initialization, and consider only the marginal work to generate a new s-dimensional quasirandom vector.

In each dimension, calculating x^{n+1} from

$$x^{n+1} = x^n \oplus v_c$$

requires the determination of c (i.e., finding the rightmost zero-bit of n) followed by a bit-by-bit exclusive-or operation. Unless you are lucky with your machine architecture, determining c takes $O(\log n)$ operations. In standard FORTRAN the bitwise exclusive-or must also be programmed. Suppose you know that at most $a - 1$ values of the sequence x^i are to be generated, and that the values x^a, x^{a+1}, ... will never be used. Then only the direction numbers $v_1, v_2, \ldots, v_b$, where $b = \lceil \log_2 a \rceil$, will be needed. Since v_i has i bits after the binary point, the necessary exclusive-or operation can be programmed as a loop over b bits, taking $O(b)$ time. Clearly $a > n$, so the whole operation in s dimensions takes $O(s\lceil \log a \rceil)$ time.

In assembly code, on the other hand, a bit-by-bit exclusive-or operation is sure to be available, taking essentially constant time for practical values of n, so the whole operation takes $O(s + \log n)$ time.

3.4 Favorable Values

As discussed in Sobol' [10, 11], a sequence of s-dimensional quasirandom vectors $q^1, q^2, \ldots, q^n$ theoretically has additional uniformity properties whenever $n = 2^k$, $k \geq \max(2s, \tau_s + s - 1)$, where τ_s is defined in [10]. The values of τ_s for some small values of s are

s	1	2	3	4	5	6	7	8	9	10	11	12	13
τ_s	0	0	1	3	5	8	11	15	19	23	27	31	35

We call these values of n *S-favorable values* (see, however, Section 4.4). Likewise, there are *F-favorable values* for Faure sequences; see Fox [4] and Section 4.4.

4. REMARKS ON THE PROGRAMS

4.1 Standard and Nonstandard FORTRAN

Standard FORTRAN [15] does not include the bit-by-bit exclusive-or operation $\oplus$. However, most machines include this operation as a single machine instruction and, reflecting this, most FORTRAN compilers also include an exclusive-or operator. In what follows, subroutines are discussed in pairs: names ending in *L* (*GOSOBL* and *INSOBL*) indicate a subroutine in standard FORTRAN, and names ending in *X* (*GOSOBX* and *INSOBX*) indicate use of a nonstandard operator available in the CDC FORTRAN compiler. To convert from the *L* to *X* version, replace all function calls of the form *EXOR*(*I*, *J*) by expressions *I .XOR. J*, where *.XOR.* is the nonstandard operator allowed by CDC; delete the *EXOR* function and all references to it.

4.2 BDSOBL

This block-data subroutine defines the primitive polynomials used to generate quasirandom vectors and provides the required initial values m_i. Since 40 polynomials are given, each with its corresponding initial values, the vectors generated can have any dimension between 1 and 40. Polynomials are defined by the array *POLY*(2:40) using a simple binary code. For instance, *POLY*(10) = 47. In binary, 47 is represented by 101111, which in turn corresponds to the primitive polynomial

$$x^5 + x^3 + x^2 + x + 1:$$

denoting the binary code by $\ldots b_2 b_1 b_0$, then $b_i = 1$ indicates that the term x^i is present in the polynomial; $b_i = 0$ indicates that x^i is absent. The polynomials defined by *POLY* are in order of increasing degree. Polynomials of the same degree are given in the order used in [13]. *POLY*(1), not defined in the program, corresponds to the special case mentioned in Section 3.1 above.

The array *MINIT*(2:40, 8) holds the initial values of m_i for each polynomial. For instance, we saw above that *POLY*(10) corresponds to a primitive polynomial of degree 5, requiring initial values for m_1 to m_5. These are found in *MINIT*(10, 1), *MINIT*(10, 2), ..., *MINIT*(10, 5). Thus for this polynomial $m_1 = 1$, $m_2 = 1$, $m_3 = 5$, $m_4 = 11$, and $m_5 = 27$. All the required initial values are taken from the table in [14]. We have not verified that they do in fact ensure properties A and A′ (see Section 3.2) as claimed by Sobol'.

This block-data subroutine includes no executable instructions. The question whether to use a nonstandard *.XOR.* operator therefore does not arise.

4.3 INSOBL and INSOBX

These subroutines initialize values needed by the Sobol' generator. The appropriate subroutine should be called exactly once for each series of points to be generated. Although the program as written allows you to generate several different series of points in the same run, it is not possible to generate two different series concurrently. Thus you should call *INSOBL* or *INSOBX* with the parameters for the first series of points, generate the first series using

GOSOBL or *GOSOBX*, call *INSOBL* or *INSOBX* with the parameters for the second series of points, generate the second series, and so on.

The parameter *FLAG* is a logical array of dimension 2. Both *FLAG*(1) and *FLAG*(2) are set *.TRUE.* if the call on *INSOBL* or *INSOBX* is correct; see below.

The integer parameter *DIMEN* gives the dimension of the quasirandom vectors to be generated. If $DIMEN < 1$ or $DIMEN > 40$, then *FLAG*(1) is set *FALSE.* to signal an error, and initialization is not completed.

The integer parameter *ATMOST* gives an upper bound on the number of points the user intends to generate in the next series to be supplied by the Sobol' generator. We saw above that the nth value x^n in a series defined by this generator is given by

$$x^n = g_1 v_1 \oplus g_2 v_2 \oplus + \cdots$$

where $\ldots g_3 g_2 g_1$ is the Gray code representation of n. Now the Gray code for n has $g_i = 0$ for $i > \lceil \log_2(n+1) \rceil$, so if we know that n will never exceed *ATMOST* we do not need to calculate v_i for $i > \lceil \log_2(ATMOST + 1) \rceil$. In our program the array V can hold values $v_1, v_2, \ldots, v_{30}$ for each of the 40 possible components of our quasirandom vectors, so *ATMOST* must not exceed $2^{30} - 1$. If this condition holds, *MAXCOL* shows which is the last column of the array V we need to initialize; otherwise *FLAG*(2) is set *.FALSE.*, and initialization is not completed.

Finally, the integer parameter *TAUS* is set by the initialization subroutine to the value of τ_s mentioned in Section 3.4.

After checking the parameters, the initialization subroutine calculates *MAXCOL* as just described and then proceeds to calculate required values of m that are not supplied in *BDSOBL*: if m_{ij} denotes the jth value of m for the ith polynomial and if this polynomial is of degree d_i, we need to calculate m_{ij} for $i = 1, 2, \ldots,$ *DIMEN* and $j = d_i + 1, d_i + 2, \ldots,$ *MAXCOL*. At this stage the array V, despite its name, is holding the values of m. To obtain the corresponding v_{ij}, we should now calculate $v_{ij} = m_{ij}/2^j$ for each i and j. It is more convenient, however, to work with an integer array, so instead the program calculates $V(I, J)$ as $m_{ij}2^{MAXCOL-j}$ and $RECIPD = (1/2)^{MAXCOL}$. Each v_{ij}, $i = 1, 2, \ldots,$ $DIMEN, j = 1, 2, \ldots, MAXCOL$ can thus be obtained as

$$v_{ij} = RECIPD * V(I, J).$$

Finally, the subroutine sets up a variable *COUNT* (which corresponds to n in our description of the generator), and initializes the vector *LASTQ*, defined in Section 4.4, to $(0, 0, \ldots, 0)$.

4.4 GOSOBL and GOSOBX

Once the generator has been initialized by a call on *INSOBL* or *INSOBX*, quasirandom vectors are generated by calling *GOSOBL* or *GOSOBX*. Each call returns a new quasirandom vector in the array *QUASI*. No additional parameters are needed, since the necessary information (the dimension of the vector, the number of the call, etc.) is saved in labelled common */SOBOL/* at initialization and thereafter between calls.

Each component of the new quasirandom vector is calculated using (**) from Section 2, where n is held in the variable *COUNT* and v_c for the ith component

of the quasirandom vector is given by $V(I, C)$. The array *LASTQ* holds the generated vectors in integer form for programming convenience, conversion to the required real values being accomplished at the last moment by multiplying by *RECIPD* (see Section 4.3).

If the user makes more calls on the generator than originally specified (using *ATMOST*, see Section 4.3), the error is trapped, and the program stops with an error message. Access to this message is implementation-dependent. It is usually printed on the standard output device.

In terms of the notation used above, the first quasirandom vector returned by these subroutines is q^1. In fact, Sobol's properties A and A′ (see Section 3.2) and the notion of favorable value (Section 3.4) are defined for sequences that begin with the quasirandom vector $q^0 = (0, 0, \ldots, 0)$. We are therefore misusing these terms to some extent. This seems preferable to including the extreme point $(0, 0, \ldots, 0)$ of the unit hypercube in the sequence of points generated.

4.5 EXOR

The (external) integer function *EXOR*(I, J), used only in the standard FORTRAN versions of the above subroutines, calculates the bit-by-bit exclusive-or of its two integer parameters.

The function exclusive-ors its parameters bit by bit starting at the least-significant end and stopping when all remaining bits of both parameters are equal. In general, therefore, the smaller the parameters, the faster the function. Inspection of *INSOBL* and *GOSOBL* will reveal that all the parameters given to *EXOR* are less than 2^{MAXCOL}, and that at least some of them are greater than or equal to $2^{MAXCOL-1}$. For efficiency, therefore, *MAXCOL*, and hence *ATMOST* (see Section 4.3), should be as small as possible, consistent with the application in hand.

When a nonstandard operator such as CDC's *.XOR.* is used, the value of *ATMOST* has no effect on the efficiency of the generator—modulo the trivial qualification that the machine can represent that value.

5. TEST RESULTS

We tested the Sobol' generator using the integral

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

discussed in [2], and previously used as a test case for other quasirandom sequence generators by Fox [4]. In that paper the Faure generator was judged to outperform the Halton and pseudorandom-sequence generators considered. The results below therefore compare only the Faure and the Sobol' generators.

5.1 Accuracy

Table II gives estimates of the above integral, rounded to three decimal places except at $N = 10^6$ where estimates are rounded to five places. The table includes

Table II. Estimates of the Integral

Generator	N	$s = 4$	$s = 7$	$s = 13$	$s = 20$	$s = 25$	$s = 40$
FAURE	500	1.000	0.982	0.658	1.047	1.048	0.193
SOBOL	500	0.976	0.961	0.741	0.442	0.788	0.235
FAURE	1000	1.011	0.996	0.993	1.054	0.863	0.208
SOBOL	1000	0.983	0.996	0.817	0.698	0.586	0.709
FAURE	7000	1.000	1.006	1.025	0.878	1.001	0.330
SOBOL	7000	0.997	0.986	0.973	0.932	0.928	0.351
FAURE	20,000	1.000	1.005	0.999	0.964	1.034	0.637
SOBOL	20,000	1.000	0.994	0.975	0.883	0.808	0.490
FAURE	40,000	1.000	1.005	0.998	0.925	0.889	0.682
SOBOL	40,000	1.000	1.001	1.010	1.013	0.908	0.459
FAURE	100,000	1.000	0.996	0.990	1.008	0.838	0.675
SOBOL	100,000	1.000	0.999	1.003	0.974	0.979	0.625
FAURE	15,625	1.000					
FAURE	78,125	1.000	*F*-favorable				
FAURE	390,625	1.000					
FAURE	16,807		1.008				
FAURE	117,649		0.998	*F*-favorable			
FAURE	823,543		1.000				
FAURE	2,197			1.000			
FAURE	28,561			1.000	*F*-favorable		
FAURE	372,293			0.992			
SOBOL	16,384	0.999					
SOBOL	65,536	1.000	*S*-favorable				
SOBOL	262,144	1.000					
SOBOL	131,072		1.000				
SOBOL	262,144		1.000	*S*-favorable			
FAURE	1,000,000	1.00000	0.99954	0.99950			
SOBOL	1,000,000	0.99999	1.00029	0.99557			

estimates at S-favorable and F-favorable values of N. In no case do we generate the point $(0, 0, \ldots, 0)$.

5.2 Speed

To compare the speed of the generators, Table III gives respective timings in seconds using the clock on the CYBER 855 computer at the University of Montreal. All programs were compiled using the FTN 5.1 compiler with the highest level of optimization ($OPT = 3$). All times include calls on the initialization subroutines (*INFAUR*, *INSOBL*, or *INSOBX*) but exclude compilation. They include not only the generation of quasirandom vectors, but also time spent estimating the integral. (These conditions are all the same, as in [4].) The timings for *SOBOL* refer to the version of this generator programmed in standard FORTRAN; timings for *SOBOX* refer to the version programmed using CDC's nonstandard *.XOR.* operator. In every case the parameter *ATMOST* was set at 100 000. These timings can be legitimately compared with those in [4] for the Halton and pseudorandom-sequence generators.

Table III. Timings

Generator	N	$s = 4$	$s = 7$	$s = 13$	$s = 20$	$s = 25$	$s = 40$
FAURE	500	0.45	0.82	1.49	2.26	2.86	4.52
SOBOL	500	0.46	0.77	1.53	2.11	2.61	4.24
SOBOX	500	0.07	0.09	0.13	0.18	0.19	0.27
FAURE	1000	0.90	1.63	2.97	4.53	5.74	9.04
SOBOL	1000	0.91	1.52	2.92	4.14	5.19	8.26
SOBOX	1000	0.13	0.17	0.24	0.33	0.38	0.52
FAURE	7000	7.22	11.2	20.5	31.5	39.7	63.2
SOBOL	7000	6.37	10.6	19.3	28.8	35.8	57.3
SOBOX	7000	0.87	1.10	1.56	2.15	2.55	3.63
FAURE	20,000	22.4	33.8	58.9	90.1	113.2	181.5
SOBOL	20,000	18.3	30.3	55.1	83.1	103.1	165.0
SOBOX	20,000	2.48	3.12	4.47	6.06	7.22	10.5
FAURE	40,000	48.2	72.3	117.6	180.8	225.6	362.3
SOBOL	40,000	36.8	60.8	110.4	167.4	207.7	331.2
SOBOX	40,000	4.95	6.26	8.95	12.2	14.4	20.9
FAURE	100,000	130.0	188.0	293.9	453.1	564.5	901.4
SOBOL	100,000	92.9	154.0	281.2	422.0	526.1	835.5
SOBOX	100,000	12.3	15.6	22.4	30.5	35.9	52.1

6. CONCLUSIONS

Based solely on these experimental results, the Faure and Sobol' generators seem roughly matched in accuracy. When implemented with a nonstandard but widely available exclusive-or function in extended FORTRAN, the Sobol' generator is much faster. If one insists on portability, the speed advantage subsists, but the difference is now small.

Fox [4] tentatively recommends the Sobol' generator for $2 \leq s \leq 6$ and the Faure generator for $s > 6$. The latter recommendation is based mainly on often-loose theoretical error bounds relevant for (possibly only impractically) large N. Hence Fox hedges.

APPENDIX A

This appendix discusses *BDSOBL, INSOBL, GOSOBL, EXOR.* These four subroutines are in standard FORTRAN 77. They are suitable for use in a program library except that, as the paper makes clear, considerable gains in efficiency can be obtained by using a nonstandard *.XOR.* operator if one is available. Subroutines corresponding to *INSOBL* and *GOSOBL* are obtained as follows: To obtain the subroutine *INSOBX*, replace the line preceding statement number 60 in *INSOBL* by

$$IF\ (INCLUD(K))\ NEWV = NEWV\ .XOR.\ (L*V(I, J - K))$$

To obtain the subroutine *GOSOBX*, replace the first line of the loop *DO* 2 in *GOSOBL* by

$$LASTQ(I) = LASTQ(I)\ .XOR.\ V(I, L)$$

APPENDIX B

This appendix discusses TESTS, which is an illustrative driver. It calls the nonstandard subroutine *SECOND* to obtain the computer clock time. Use a local counterpart to this subroutine if you have one; otherwise delete all references to *SECOND*, *T1*, and *T2*. Input and output unit numbers are specified in a *PARAMETER* statement. Alter this as required by your local operating system. We use free-format input and output throughout for convenience. The portable subroutine *GOSOBL*, or its machine-dependent counterpart *GOSOBX*, may in certain circumstances stop the program with an appropriate error message (produced using the standard FORTRAN STOP statement: see [15, Section 11.12]). All other output comes from the driver program. Sample input values to *TESTS* are *DIMEN* = 7 and *ATMOST* = 1000. If your machine has words with at least 32 bits and a bitwise exclusive-or in its extended FORTRAN, you can adapt *GOSOBX* by replacing *.XOR.* by its local analog as indicated above.

Listings of the driver program and the subroutines used to test the Faure generator are given in the preprint of [4] and are available in the CALGO listings.

ACKNOWLEDGMENTS

We thank I. M. Sobol' for sending copies of references [13] and [14] and P. K. Sarkar for sending a copy of reference [9]. We also thank an anonymous referee whose suggestions considerably improved our programs.

REFERENCES

1. ANTONOV, I. A., AND SALEEV, V. M. An economic method of computing LP_τ-sequences. *USSR Comput. Math. Math. Phys. 19* (1979), 252–256.
2. DAVIS, P. J., AND RABINOWITZ, P. *Methods of Numerical Integration.* Academic Press, New York, 1983.
3. FAURE, H. Discrépance de suites associées à un système de numération (en dimension s). *Acta Arithmetica XLI* (1982), 337–351.
4. FOX, B. L. ACM Algorithm 647. Implementation and relative efficiency of quasirandom sequence generators. *ACM Trans. Math. Softw. 12* (1986), 362–376.
5. HALTON, J. H. On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals. *Numer. Math. 2* (1960), 84–90.
6. KNUTH, D. E. *The Art of Computer Programming, vol. 2: Seminumerical Algorithms.* 2nd ed., Addison-Wesley, Reading, Mass., 1981.
7. PETERSON, W. W., AND WELDON, E. J. *Error-Correcting Codes.* 2nd ed. MIT Press, Cambridge, Mass., 1972.
8. NIEDERREITER, H. Point sets and sequences with small discrepancy. *Monatsh. Math.* (to appear).
9. SARKAR, P. K., AND PRASAD, M. A. A comparative study of pseudo and quasirandom sequences for the solution of integral equations. *J. Comput. Phys. 68* (1987), 66–88.
10. SOBOL', I. M. On the distribution of points in a cube and the approximate evaluation of integrals. *USSR Comput. Math. Math. Phys. 7* (1967), 86–112.
11. SOBOL', I. M. Uniformly distributed sequences with an additional uniform property. *USSR Comput. Math. Math. Phys. 16* (1976), 236–242.
12. SOBOL', I. M. On the systematic search in a hypercube. *SIAM J. Numer. Anal. 16* (1979), 790–793.
13. SOBOL', I. M. Points which uniformly fill a multidimensional cube. *Math. Cybern. 2* (1985) Znanie, Moscow (in Russian).

14. SOBOL', I. M., AND LEVITAN, YU.L. The production of points uniformly distributed in a multidimensional cube. Preprint 40, Institute of Applied Mathematics, USSR Academy of Sciences, 1976 (in Russian).
15. AMERICAN NATIONAL STANDARDS INSTITUTES. *American National Standard Programming Language FORTRAN*. Standard X3.9-1978, American National Standards Institute, New York, 1978.

Received November 1986; revised July and October 1987; accepted November 1987