An Efficient Algorithm for Sequential Random Sampling

JEFFREY SCOTT VITTER
I.N.R.I.A. Brown University

We examine several methods for drawing a sequential random sample of n records from a file containing N records. Method D is recommended for general use. The algorithm is on-line (so that CPU time can be overlapped with I/O), has a small constant memory requirement, and is easy to program. An improved implementation is detailed in the Appendix.

Categories and Subject Descriptors: F.2.m [**Analysis of Algorithms and Problem Complexity**]: Miscellaneous; G.3 [**Mathematics of Computing**]: Probability and Statistics—*random number generation*; *statistical software*

General Terms: Algorithms, Design, Performance, Theory

Additional Key Words and Phrases: Optimization, random sampling, rejection method

1. INTRODUCTION

Sequential random sampling is a fundamental operation having many applications in science and industry. The problem is to draw a random sample of size n without replacement from a file containing N records; the n records must appear in the same order in the sample as they do in the file. Another formulation is to form a sorted random set of n elements from $\{1, 2, \ldots, N\}$. The sample size n is typically very small relative to the file size N.

In this paper we reaffirm the efficiency and practicality of Method D introduced in [9] and present some implementation improvements. We recommend it as the method of choice for sequential random sampling. The main features of Method D are

(1) It is *on-line*—that is, it requires no preprocessing and can generate each element of the sample in constant expected time.
(2) The memory requirement of the program is a small constant.
(3) The implementation is easy to program (it is given in the Appendix).

Method D solves the open problem presented in [8, ex. 3.4.2–8].

Support was provided in part by a National Science Foundation research grant DCR-84-03613, by an NSF Presidential Young Investigator Award with matching funds from an IBM Faculty Development Award and an AT&T research grant, and by a Guggenheim Fellowship.
Author's address: Department of Computer Science, Brown University, Providence, R.I. 02912. Part of this research was done while the author was on leave at the Mathematical Sciences Research Institute in Berkeley, California, and the article was revised while on sabbatical at Institut National de Recherche en Informatique et en Automatique in Rocquencourt, France, and at the École Normale Supérieure in Paris, France.

Two interesting twopass methods for sequential random sampling, called Methods SG and SG*, were developed by Ahrens and Dieter [2]. Methods SG and SG* were quoted in [2] to be faster than Method D by factors of 4–5 and 2–2.5, respectively. It turns out that an inferior implementation of Method D was used in the testing, not the one given explicitly on page 716 in [9]. We have redone the experiments with a better implementation. Method D uses more CPU time than Method SG in most cases, but always less than Method SG*.

In terms of elapsed time, Method D is the fastest way to obtain a random sample of records because its CPU time can be overlapped with the record I/O, whereas Methods SG and SG* use two passes and are not on-line. Method SG also requires $O(n)$ memory space. When n is large, this can be prohibitive, and in virtual memory environments page faults can deteriorate performance.

The next section gives a brief description of Method D along with the theory behind the implementation in the Appendix. In Section 3, we review the structure of Methods SG and SG* and present our empirical CPU timings. The algorithms were coded carefully in FORTRAN 77 (VS FORTRAN) on an IBM 3081 mainframe. All three methods can be sped up further by using an assembly-coded exponential random number generator, if available. Conclusions are given in Section 4. Efficient and improved implementations of Method D and another method, called Method A, appear in the Appendix.

2. METHOD D

We begin by discussing the simplest of all sequential random sampling methods, developed by Fan, Muller, and Rezucha [4] and Jones [6], which is called Method S by Knuth [9]. An independent uniform random variate (from the unit interval) is generated for each record in the file in order to determine whether the record should be chosen for the sample. If m records have already been chosen from among the first t records in the file, the $(t + 1)$st record is chosen with probability $(n - m)/(N - t)$. Method S thus requires roughly N random variates and runs in $O(N)$ time.

Significant speedups can be obtained by determining in an efficient way how many records should be *skipped over* before the next is chosen for the sample. Let us denote the number of records to skip by $S(n, N)$, the *skip distance*. We also modify our definitions of n and N to be, respectively, the number of records remaining to be chosen for the sample and the number of records in the file that have not yet been processed.

The range of $S(n, N)$ is the integers in $[0, N - n]$. The distribution function $F(s) = \Pr\{S \le s\}$ is

$$F(s) = 1 - \frac{(N - s - 1)^{\underline{n}}}{N^{\underline{n}}} = 1 - \frac{(N - n)^{\underline{s+1}}}{N^{\underline{s+1}}}, \tag{2.1}$$

for $0 \le s \le N - n$. (The notation $a^{\underline{b}}$ denotes the *falling power* $a(a - 1) \ldots (a - b + 1) = a!/(a - b)!$) It follows that the probability function $f(s) = \Pr\{S = s\}$ is

$$f(s) = \frac{n}{N} \frac{(N - s - 1)^{\underline{n-1}}}{(N - 1)^{\underline{n-1}}} = \frac{n(N - n)^{\underline{s}}}{N(N - 1)^{\underline{s}}}, \tag{2.2}$$

for $0 \le s \le N - n$. The mean value of $S(n, N)$ is $(N - n)/(n + 1)$, and the standard deviation is roughly equal to the mean. The full derivations of most of the results presented in this section appear in [9].

Method S can be expressed in this framework: Each value of the skip distance $S(n, N)$ is generated in $O(S + 1)$ time, using $S + 1$ uniform random variates. The running time can be improved by a factor of 3–4 if we determine S by generating a uniform random variate V and setting S to be the minimum value $s \ge 0$, such that $(N - n)^{\underline{s+1}} \le N^{\underline{s+1}} V$. It follows from (2.1) that the resulting S has the correct distribution. The running time is still $O(N)$, but only n random variates are generated. An efficient implementation for this algorithm, which we call Method A, is given in the Appendix.

Method D achieves its greater speedup by generating S using the acceptance–rejection framework of von Neumann. The trick is to find a random variable X with probability density function $g(x)$ that approximates S well and a constant c such that $f(\lfloor x \rfloor) \le cg(x)$, for all x in the domain of $g(x)$. To generate S, we independently generate X and a uniform random variate U. If $U \le f(\lfloor X \rfloor)/cg(X)$ (which occurs with very high probability), then we *accept* and set $S := \lfloor X \rfloor$; otherwise we *reject* and repeat the process with a new X and U. The resulting S has the correct distribution (2.1).

The computation of $f(s)$ is expensive and requires $\theta(\min\{n, s\})$ time because of the presence of the falling powers in (2.2). Instead, we use a quickly computed approximation $h(s)$ satisfying $h(s) \le f(s)$. The test $U \le h(\lfloor X \rfloor)/cg(X)$ is performed instead, and, if satisfied, we accept and set $S := \lfloor X \rfloor$, since it then follows that $U \le f(\lfloor X \rfloor)/cg(X)$. Otherwise, the test $U \le f(\lfloor X \rfloor)/cg(X)$ is performed, and we accept or reject as before. The process is repeated until acceptance is reached. This technique is sometimes called a *squeeze method* since the value of $f(\lfloor x \rfloor)$ is squeezed between the values $h(\lfloor x \rfloor)$ and $cg(x)$.

The algorithm is very short and simple to state. The parameter α determines when to use Method A as opposed to the acceptance–rejection technique. Typical settings are in the range 0.05–0.15. For the efficient implementation given in the Appendix, we use $\alpha = \frac{1}{13}$. The algorithm works as follows:

D1. [Is $n \ge \alpha N$?] If $n \ge \alpha N$, use Method A to finish the sampling and then terminate.

D2. [Generate U and X.] Generate independently a uniform random variate U from the unit interval and a random variate X with density function $g(x)$. If $X \ge N - n + 1$, then X is regenerated until $X < N - n + 1$.

D3. [Accept?] If $U \le h(\lfloor X \rfloor)/cg(X)$, then set $S := \lfloor X \rfloor$ and go to Step D5.

D4. [Accept?] If $U \le f(\lfloor X \rfloor)/cg(X)$, then set $S := \lfloor X \rfloor$. Otherwise, return to Step D2.

D5. [Select the $(S + 1)$st record.] Skip over the next S records in the file and then select the following one for the sample. Set $N := N - S - 1$ and $n := n - 1$. Return to Step D1 if $n > 0$.

A good choice for X is the beta distribution scaled to the interval $[0, N]$ with parameters $a = 1$, $b = n$. It is the continuous counterpart of S and approximates S very well. The value of X can be thought of as the minimum of n numbers chosen independently and uniformly from the real interval $[0, N]$. We can generate X by setting

$$X := N(1 - U^{1/n}) \qquad \text{or} \qquad X := N(1 - e^{-Y/n}), \tag{2.3}$$

where U is uniformly distributed on the unit interval and Y is exponentially distributed. (Expressions of the form a^b are evaluated as $\exp(b \ln a)$ using a call to the exponential and logarithm library functions.)

Our choice of parameters is given below:

$$g(x) = \begin{cases} \dfrac{n}{N}\left(1 - \dfrac{x}{N}\right)^{n-1}, & 0 \le x \le N; \\ 0, & \text{otherwise;} \end{cases}$$

$$c = \frac{N}{N - n + 1}; \tag{2.4}$$

$$h(s) = \begin{cases} \dfrac{n}{N}\left(1 - \dfrac{s}{N - n + 1}\right)^{n-1}, & 0 \le s \le N - n; \\ 0, & \text{otherwise.} \end{cases}$$

The resulting algorithm is proved in [9] to run in $O(n)$ time, on the average.

The running time can be cut by more than half by clever implementation. For our choice of parameters in (2.4), the test of acceptance in Step D3 is of the form

$$\text{Is} \quad U \le \frac{N - n + 1}{N}\left(\frac{N - n - S + 1}{N - n + 1} \cdot \frac{N}{N - X}\right)^{n-1} ? \tag{2.5}$$

When the test is true, which happens with very high probability $1 - O(n/N)$, the ratio of the left-hand side and the right-hand side of (2.5), namely,

$$\frac{NU}{N - n + 1}\left(\frac{N - n + 1}{N - n - S + 1} \cdot \frac{N - X}{N}\right)^{n-1}, \tag{2.6}$$

is statistically equivalent to a uniform random variate whose value is independent of all previous values of X and of whether those X were accepted. We get the $(n - 1)$st root of a uniform random variate by taking the $(n - 1)$st root of (2.6). Let us denote the resulting quantity by V':

$$V' = \left(\frac{NU}{N - n + 1}\right)^{1/(n-1)} \frac{N - n + 1}{N - n - S + 1} \cdot \frac{N - X}{N}. \tag{2.7}$$

A more efficient way to order these steps is first to compute V' via (2.7) and then to do the simple test of acceptance

$$\text{Is} \quad V' \le 1? \tag{2.8}$$

which is equivalent to (2.5). If the test (2.8) is satisfied, then we can efficiently precompute the value of X for the next iteration of the algorithm by setting

$$X := N(1 - V'). \tag{2.9}$$

The reader should compare this with (2.3) and note that n decreases by 1 before the start of the next iteration. We thus obtain the next X without need of a uniform random variate and an operation of the form a^b, which is normally suggested by (2.3). The resulting X has the necessary independence, as stated above. Another nice feature of the test (2.8) is that the possibility of floating point underflow is eliminated. With this implementation, each of the n iterations

of Method D requires an average of about one uniform random variate and one computation of the form a^b (which is implemented as $\exp(b \ln a)$). This represents a twofold speedup in running time.

An alternate way to generate X, as suggested by (2.3), is to use an exponentially distributed random variate Y, which has the same distribution as $-\ln U$. There are assembly language programs for generating Y more quickly than by computing the logarithm of a uniform random variate. This fact can be incorporated into the above scheme to reduce the overhead per iteration to about one exponential random variate and one exponential operation of the form e^a. The fix is not straightforward, however, because the argument to the logarithm function implicit in (2.7) is $cU = NU/(N - n + 1)$, not U. The solution is to redefine c to be

$$c = \exp\left(\frac{n-1}{N-n+1}\right). \tag{2.10}$$

The definition of V' changes from (2.7) to

$$V' = \exp\left(\frac{1}{N-n+1} - \frac{Y}{n-1}\right) \cdot \frac{N-n+1}{N-n-S+1} \cdot \frac{N-X}{N}. \tag{2.11}$$

If the test $V' \le 1$, is true, which again happens with very high probability $1 - O(n/N)$, then we generate X for the next loop via (2.9) as before. The resulting implementation is given in the Appendix.

Two optimizations of a relatively minor nature are possible in the way that X is generated. When $1/n$ is sufficiently small with respect to n/N, the average number of times that Step D2 is done before acceptance is reached can be reduced by using a different distribution for X, namely, the geometric distribution given in [9]. The second optimization comes into play when n is sufficiently small and when X cannot be generated via (2.9). An alternative to (2.3) is to obtain X directly by generating n independent uniform real numbers from the unit interval, selecting the smallest, and multiplying it by N. But since (2.3) is rarely used to generate X (the significantly faster (2.9) is used whenever possible), the speedup is minimal. In each case, *sufficiently small* means smaller than some implementation-dependent constant. The improvements are minor, at the cost of more complicated code, and for those reasons are not included in the Appendix. Another possible optimization is to reduce the overhead in calling the random number generator by generating several random numbers at once and storing them in an array. This requires some extra storage, and the program goes off-line from time to time to do the random number generation.

3. COMPARISONS

In this section we compare the performance of Method D with that of the twopass random sampling schemes Methods SG and SG* from [2]. The latter two algorithms are not on-line, but do run in $O(n)$ time. Let us begin with a sketch of how Methods SG and SG* work. In the first pass, geometrically distributed random variates $G_1, G_2, \ldots$ with a fixed mean are used as the skip distances. If less than n records are selected, that is, if $\sum_{1 \le i \le n}(G_i + 1) > N$, then the process is repeated. The mean μ of the geometric random variates is chosen slightly less than $N/n - 1$ so that the odds of repeating are small. Typically no I/O is done during this pass. Each time a record is "selected" for the sample its index is

recorded in an array, which requires space for $O(n)$ pointers. If array space runs out, the process must be restarted. In the second pass, the number of elements in the array is reduced to n by randomly deleting elements. The n entries are then compacted, and the actual selection of the records can begin. (An alternative is for the records to be actually selected in the first pass and stored in internal memory or written to secondary storage; the extraneous ones would be deleted in pass two.)

Method SG has a memory storage requirement of $O(n)$, which can be excessive. This storage requirement can be avoided if the random number generator used to generate $G_1, G_2, \ldots$ can be reseeded for the second pass, so that the program can regenerate the selected indices on the fly. The final n indices are then chosen via Method S (or better yet, via Method A) using a different sequence of pseudorandom numbers. The resulting algorithm is called Method SG*.

Methods SG and SG* are very similar to the clever algorithms proposed by J. L. Bentley (personal communication, April 1983; see [9, p. 713]). The main difference is that Bentley obtains a sorted random sample of real numbers by repeatedly generating random variates X using (2.3); the parameters n and N change dynamically as the sampling progresses. The sample of reals is then truncated to integers to complete pass one. The rest of the algorithms are the same as Methods SG and SG*. The size of the sample formed in pass one by Bentley's methods has a much smaller variance than the size of the sample produced by Methods SG and SG*, so the target mean size of the sample can be reduced. However, it is more costly to generate random variates X than geometrically distributed random variates with a fixed mean, so Methods SG and SG* are respectively faster than the corresponding methods due to Bentley.

The CPU times (in microseconds) per selected record for Methods D, SG, and SG* are given in Table I for $N = 10^6$ and in Table II for $N = 10^8$. Each algorithm was simulated $10^5/n$ times for each value of n and N, and the cumulated CPU time was divided by 10^5 to get the average time per selected record. An exception was the case $n = 10^6$, $N = 10^8$, which was simulated once for each algorithm; the running time was then divided by n to get the corresponding table entry.

The programs were implemented using FORTRAN 77 (VS FORTRAN), and special care was taken to avoid conversions between different data formats (like *integer* and *real*) which slow down the running time. The programs were compiled and run on an IBM 3081 mainframe computer; optimization level one was used, since it produced the fastest code. A Pascal-like implementation of Method D is given in the Appendix; the FORTRAN 77 version is a direct translation of it.

The comparisons given in [2] involving Method D are misleading because the timing experiments use the inferior implementation of Method D in [5], not the simpler and more efficient version given on page 716 in [9] or the improved version given in the Appendix of this paper. Methods S and A, which are not included in the above tables, use roughly $12N$ and $3.5N + 8n$ microseconds of CPU time, respectively, to complete the sampling.

In terms of elapsed time, Method D is the fastest way to generate a sequential random sample of records, since it is an on-line algorithm and its CPU computation can be overlapped with the I/O; that is not the case with Methods SG and SG*. Even if the records are actually selected during pass one and stored on secondary storage for pass two, the I/O time in the second pass will not be overlapped and will typically be greater than the CPU time. In addition,

Table I. CPU Time in Microseconds for $N = 10^6$

$N = 10^6$	D	SG	SG*
$n = 1$	39	146	175
$n = 2$	42	98	133
$n = 5$	44	67	103
$n = 10^1$	44	53	90
$n = 10^2$	45	35	69
$n = 10^3$	48	29	62
$n = 10^4$	54	27	59
$n = 10^5$	44	26	58

Table II. CPU Time in Microseconds for $N = 10^8$

$N = 10^8$	D	SG	SG*
$n = 1$	39	146	172
$n = 2$	42	98	131
$n = 5$	44	67	103
$n = 10^1$	44	53	90
$n = 10^2$	45	35	69
$n = 10^3$	45	29	62
$n = 10^4$	48	27	59
$n = 10^5$	55	26	58
$n = 10^6$	44	26	58

Method SG requires $O(n)$ space, which can be excessive and can cause page faults when n is large.

A useful and intuitive measure of CPU performance is the number of mathematical library function calls. For simplicity, we shall ignore arithmetic operations like addition, subtraction, multiplication, and division, and instead concentrate on the number of random variates generated and the number of exponential and logarithm computations. In Methods SG and SG*, the geometric random variates each have the same mean (call it μ), so they can each be generated by one uniform random variate and one logarithm operation (or equivalently, by one exponential random variate). Method SG requires the computation of an average of about $N/(\mu + 1) = n + O(n^{1/2})$ geometric random variates in the first pass and about $N/(\mu + 1) - n = O(n^{1/2})$ uniform random variates in the second pass. Method SG* requires an average of roughly $n + O(n^{1/2})$ geometric random variates in each of the two passes, plus an extra $n + O(n^{1/2})$ uniform random variates in the second pass. Method D uses an average of about n uniform random variates, n exponential operations, and n logarithm operations (or equivalently, about n exponential random variates and n exponential operations).

It is possible to generate an exponential random variate faster than by computing $-\ln U$, where U is a uniform random variate, through direct techniques coded in assembly language [1, 8]. If such a routine is available, all three methods can be sped up. The way to do that for Method D is described in the Appendix.

There are several published nonsequential random sampling methods (e.g., see [2, 3, 8]) that typically use a hash table of size $O(n)$. Sequential samples can be

obtained in a second pass by sorting, also in $O(n)$ time. The CPU times are similar to those of Method SG; the same disadvantages apply, except that problems due to page faults are even more likely.

4. CONCLUSIONS

Method D is the recommended algorithm for sequential random sampling. It combines the advantages of Method A when n is large with the efficiency of the acceptance–rejection technique for small n. The method is on-line, and thus its CPU time can be overlapped with the I/O when records are selected. The memory requirement is constant. An efficient implementation is given in the Appendix. CPU time using FORTRAN 77 on an IBM 3081 is $\approx$ 40–50 microseconds per selected record; this can be improved further if a fast exponential random variate generator is available.

The Appendix also gives an efficient version of Method A, which improves the running time of the basic Method S by a factor of 3–4. Other algorithms for sequential random sampling are described and analyzed in [7] and [9].

APPENDIX

In this appendix we present Pascal-like implementations for Methods A and D. Variables of type *real* should be double precision so that roundoff error is insignificant, even when N is very large. Roughly $\log_{10} N + 1$ digits of precision will suffice. Intermediate calculations should be done in double precision as well. Variables of type *integer* must be able to store values up to size N. The function call *UNIFORMRV*( ) returns a uniform random variable from the open unit interval. The reserved words **loop** and **endloop** denote the beginning and end of a loop. The statement **breakloop** causes flow of control to exit the innermost active loop. We assume that $n \geq 1$. The seemingly redundant type conversions for the special case $n = 1$ in each algorithm are to ensure that the out-of-range value $S = N$ is never generated because of roundoff error. Such a situation has never been observed by the author, but it is theoretically possible, so the precaution is warranted. The extra time required is insignificant.

```
top := N − n; Nreal := N;
while n ≥ 2 do begin
    V := UNIFORMRV( ); S := 0; quot := top/Nreal;
    while quot > V do begin
        S := S + 1; top := −1.0 + top; Nreal := −1.0 + Nreal;
        quot := (quot × top)/Nreal
        end;
    Skip over the next S records and select the following one for the sample;
    Nreal := −1.0 + Nreal; n := −1 + n
    end;
{ Special case n = 1 }
S := TRUNC(ROUND(Nreal) × UNIFORMRV( ));
Skip over the next S records and select the following one for the sample;
```

A1. Method A. Variables *V*, *quot*, *Nreal*, and *top* are of type *real*. All other variables have type *integer*. The variable *Nreal* represents N in floating point format.

A2. Method D. We use an *integer* variable *negalphainv* to represent $-1/\alpha$ where α is the parameter that decides when to use Algorithm A instead of the acceptance–rejection method. Typical values of α are in the range 0.05–0.15. For example, in the IBM 3081 implementation, we used $\alpha = \frac{1}{13}$.

The variables *nreal*, *Nreal*, *ninv*, *nmin1inv*, *U*, *X*, *Vprime*, *y1*, *y2*, *top*, *bottom*, *negSreal*, and *qu1real* have type *real*; the others have type *integer*. Variables *nreal*, *Nreal*, *ninv*, *nmin1inv*, and *negSreal* are floating point versions of n, N, $1/n$, $1/(n-1)$, and $-S$, respectively. The value of *qu1* is $N - n + 1$, and *qu1real* is its floating point representation. Variable *Vprime* is equal to the nth root of a uniform random variable when it is used to generate X via (2.9).

```
nreal := n; ninv := 1.0/nreal; Nreal := N;
Vprime := EXP(LOG(UNIFORMRV( )) × ninv);
qu1 := −n + 1 + N; qu1real := −nreal + 1.0 + Nreal;
negalphainv := −13; threshold := −negalphainv × n;
while (n > 1) and (threshold < N) do
    begin
    nmin1inv := 1.0/(−1.0 + nreal);
    loop
        loop  { Step D2: Generate U and X }
            X := Nreal × (− Vprime + 1.0); S := TRUNC(X);
            if S < qu1 then breakloop;
            Vprime := EXP(LOG(UNIFORMRV( )) × ninv)
        endloop;
        U := UNIFORMRV( ); negSreal := −S;
        { Step D3: Accept? }
        y1 := EXP(LOG(U × Nreal/qu1real) × nmin1inv);
        Vprime := y1 × (−X/Nreal + 1.0) × (qu1real/(negSreal + qu1real));
        if Vprime ≤ 1.0 then breakloop;  { Accept! Test (2.8) is true }
        { Step D4: Accept? }
        y2 := 1.0; top := −1.0 + Nreal;
        if −1 + n > S then
            begin bottom := −nreal + Nreal; limit := −S + N end
        else begin bottom := −1.0 + negSreal + Nreal; limit := qu1 end;
        for t := −1 + N downto limit do begin
            y2 := (y2 × top)/bottom;
            top := −1.0 + top; bottom := −1.0 + bottom end;
        if Nreal/(−X + Nreal) ≥ y1 × EXP(LOG(y2) × nmin1inv) then
            begin  { Accept! }
            Vprime := EXP(LOG(UNIFORMRV( )) × nmin1inv);
            breakloop
            end;
        Vprime := EXP(LOG(UNIFORMRV( )) × ninv)
    endloop;
    { Step D5: Select the (S + 1)st record }
    Skip over the next S records and select the following one for the sample;
    N := −S + (−1 + N); Nreal := negSreal + (−1.0 + Nreal);
    n := −1 + n; nreal := −1.0 + nreal; ninv := nmin1inv;
    qu1 := −S + qu1; qu1real := negSreal + qu1real;
    threshold = threshold + negalphainv
    end;
if n > 1 then Use Method A to finish the sampling
else begin  { Special case n = 1 }
    S := TRUNC(N × Vprime);
    Skip over the next S records and select the following one for the sample
    end;
```

When the test (2.8) is true, which happens with very high probability $1 - O(n/N)$, the random variable X in the next iteration is generated via (2.9). This is significantly faster than (2.3), since it saves a call to *UNIFORMRV*, *EXP*, and *LOG*.

If a fast assembly language subroutine is available for generating exponentially distributed random variates, then we can speed up the algorithm, as outlined in (2.10) and (2.11). The two lines of code

```
U := UNIFORMRV( );
y1 := EXP(LOG(U × Nreal/qu1real) × nmin1inv);
```

in Steps D2 and D3 should be replaced by

```
Y := EXPONENTIALRV( );
y1 := EXP(−Y × nmin1inv + 1.0/qu1real);
```

and all expressions of the form *LOG*(*UNIFORMRV*( )) in the code should be replaced by −*EXPONENTIALRV*( ).

ACKNOWLEDGMENTS

The author would like to thank the referees for their very helpful comments.

REFERENCES

1. AHRENS, J. H., AND DIETER, U. Computer methods for sampling from the exponential and normal distributions. *Commun. ACM 15*, 10 (Oct. 1972), 873–882.
2. AHRENS, J. H., AND DIETER, U. Sequential random sampling. *ACM Trans. Math. Softw. 11*, 2 (June 1985) 157–169.
3. ERNVALL, J., AND NEVALAINEN, O. An algorithm for unbiased random sampling. *Comput. J. 25*, 1 (Jan. 1982) 45–47.
4. FAN, C. T., MULLER, M. E., AND REZUCHA, I. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *Am. Stat. Assn. J. 57* (June 1962) 387–402.
5. GEHRKE, H. Einfache sequentielle Stichprobenentnahme. Diplomarbeit, Universität Kiel, Kiel, West Germany (Aug. 1984).
6. JONES, T. G. A note on sampling a tape file. *Commun. ACM 5*, 6 (June 1962) 343.
7. KAWARASAKI, J., AND SIBUYA, M. Random numbers for simple random sampling without replacement. *Keio Math. Sem. Rep. 7* (1982) 1–9.
8. KNUTH, D. E. *The Art of Computer Programming. Vol. 2, Seminumerical Algorithms*, 2d ed. Addison-Wesley, Reading, Mass. (1981).
9. VITTER, J. S. Faster methods for random sampling. *Commun. ACM 27*, 7 (July 1984) 703–718.

Received July 1986; revised December 1986; accepted February 1987