INTERNATIONAL STANDARD

ISO 28640

First edition
2010-03-15

Random variate generation methods

Méthodes de génération de nombres pseudo-aléatoires

Reference number
ISO 28640:2010(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO 28640 was prepared by Technical Committee ISO/TC 69, *Applications of statistical methods*.

This is the first edition.

Introduction

This International Standard specifies typical algorithms by which the users can regard the generated numerical sequences as if they were real random variates.

Nowadays most statisticians, scientists and engineers have enough computer power at their disposal to carry out large computer simulations, and it is important that these be based on sound pseudo-random generators. This International Standard has been developed to help ensure that randomization, where needed, is carried out correctly and efficiently.

Six uses of randomization can be identified in statistical standardization:

— selection of a random sample;

— analysis of sample data;

— development of standards;

— checking theoretical results;

— demonstrating that a proposed procedure has the properties claimed of it;

— resolving uncertainty in the statistical literature.

Random variate generation methods

1 Scope

This International Standard specifies methods for generating uniform and non-uniform random variates for Monte Carlo simulation purposes. Cryptographic random number generation methods are not included. This International Standard is applicable, *inter alia*, by

— researchers, industrial engineers or experts in operations management, who use statistical simulation,

— statisticians who need randomization related to SQC methods, statistical design of experiments or sample surveys,

— applied mathematicians who plan complex optimization procedures that require the use of Monte Carlo methods, and

— software engineers who implement algorithms for random variate generation.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 2382-1, *Information technology — Vocabulary — Part 1: Fundamental terms*

ISO 3534-1, *Statistics — Vocabulary and symbols — Part 1: General statistical terms and terms used in probability*

ISO 3534-2, *Statistics — Vocabulary and symbols — Part 2: Applied statistics*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382-1, ISO 3534-1 and ISO 3534-2 apply, except where redefined below.

3.1
random variate
random number
number as the realization of a specific random variable

NOTE 1 The term "random number" is often used for uniformly distributed random variate.

NOTE 2 Random numbers provided as a sequence are called a "random number sequence".

3.2
pseudo-random number
random number (3.1) generated by an algorithm, that appears to be random

NOTE If there is no fear of misunderstanding, a pseudo-random number may simply be called a “random number”.

3.3
physical random number
random number (3.1) generated by a physical mechanism

3.4
binary random number sequence
random number (3.1) sequence consisting of zeros and ones

3.5
seed
initialization value required for pseudo-random number generation

4 Symbols and mathematical binary operations

4.1 Symbols

For the purposes of this document, the symbols given in the normative references as the latest versions of ISO/IEC 2382-1, ISO 3534-1 and ISO 3534-2 apply, except where redefined below.

The symbols and abbreviations specifically used in this International Standard are as follows:

X	integer type uniform random number
U	standard uniform random number
Z	normal random variate
n	suffix of random number sequence

4.2 Mathematical binary operations

The mathematical binary operations specifically used in this International Standard are as follows:

$\mathrm{mod}(m; k)$	residue from dividing integer m by k
$m \oplus k$	bitwise exclusive logical disjunction of binary integers m and k

EXAMPLE 1 $1 \oplus 1 = 0$

$0 \oplus 1 = 1$

$1 \oplus 0 = 1$

$0 \oplus 0 = 0$

$1010 \oplus 1100 = 0110$

$m \wedge k$	bitwise logical conjunction of binary integers m and k
EXAMPLE 2	$1 \wedge 1 = 1$
	$0 \wedge 1 = 0$
	$1 \wedge 0 = 0$
	$0 \wedge 0 = 0$
	$1010 \wedge 1100 = 1000$
$m := k$	replaces value m by k
$m >> k$	k-bit right shift of binary integer m
$m << k$	k-bit left shift of binary integer m

5 Uniformly distributed pseudo-random numbers

5.1 General

This clause provides algorithms for generating uniformly distributed pseudo-random numbers based on M-sequence methods (see 5.2).

Annex A introduces the concept of physically generated random numbers for information.

Annex B includes C and full Basic codes for all the recommended algorithms for information. Although the linear congruential method is not recommended for complex Monte Carlo simulations, it is also included in Annex B for information.

5.2 M-sequence method definition

a) Let p be a natural number, and c_1, c_2, ..., c_{p-1} be specified to be 0 or 1, and define the recurrence formula

$$x_{n+p} = c_{p-1}\,x_{n+p-1} + c_{p-2}\,x_{n+p-2} + \ldots + c_1\,x_{n+1} + x_n \pmod 2 \quad (n = 1, 2, 3, \ldots)$$

b) The least positive integer N such that $x_{n+N} = x_n$ for all n is called the period of the sequence. This sequence is called an M-sequence in cases where its period is $2^p - 1$.

c) The polynomial

$$t^p + c_{p-1}\,t^{p-1} + \ldots + c_1 t + 1$$

is called the characteristic polynomial of the above-mentioned recurrence formula.

NOTE 1 A necessary and sufficient condition for the above-mentioned recurrence formula to generate an M-sequence is that at least one of the seeds x_1, x_2, ..., x_p is not zero.

NOTE 2 The letter M of the M-sequence originates from the English word “maximum”, which means the largest. The period of any sequence generated by the above recurrence formula cannot exceed $2^p - 1$. Therefore, if there is a series that has a period of $2^p - 1$, it is the series that has the largest period.

NOTE 3 When this method is used, either one of the polynomials listed in Table 1 or another primitive polynomial listed in the literature is chosen as the characteristic polynomial and its coefficients are used to define the recurrence formula in a).

5.3 Pentanomial GFSR method

This method uses a characteristic polynomial of 5 terms, and it generates binary integer sequences of w bits by the following recurrence formula. This algorithm is called the GFSR or “generalized feedback shift register” random number generator.

$$X_{n+p} = X_{n+q1} \oplus X_{n+q2} \oplus X_{n+q3} \oplus X_n \quad (n = 1, 2, 3, ...)$$

The parameters are (p, q1, q2, q3, w) and X_1, ..., X_p are initially given as seeds. Examples of parameters p, q_1, q_2, q_3 giving the largest period $2^p - 1$ are indicated in Table 1.

Table 1 — Pentanomial characteristic polynomials

p	q_1	q_2	q_3
89	20	40	69
107	31	57	82
127	22	63	83
521	86	197	447
607	167	307	461
1 279	339	630	988
2 203	585	1 197	1 656
2 281	577	1 109	1 709
3 217	809	1 621	2 381
4 253	1 093	2 254	3 297
4 423	1 171	2 273	3 299
9 689	2 799	5 463	7 712
NOTE q_1, q_2, q_3 represent exponents of the non-zero terms of the characteristic polynomial.			

5.4 Combined Tausworthe method

Let x_0, x_1, x_2, … be an M-sequence generated by the recurrence relationship:

$$x_{n+p} = x_{n+q} + x_n \text{ (mod 2)} \quad (n = 0, 1, 2, \ldots)$$

Using this M-sequence, a w-bit integer sequence called a simple Tausworthe sequence with parameters (p, q, t) is obtained as follows:

$$X_n = x_{nt}x_{nt+1}\ldots x_{nt+w-1} \quad (n = 0, 1, 2, \ldots)$$

where

t is a natural number which is coprime to the period $2^p - 1$ of the M-sequence;

w is the word length which does not exceed p.

The period of this sequence is also $2^p - 1$.

NOTE 1 Two integers are said to be coprime, or relatively prime, when they have no common divisors other than unity.

EXAMPLE If a primitive polynomial $t^4 + t + 1$ is chosen, set $p = 4$, and $q = 1$ in the above recurrence relationship. If the seeds $(x_0, x_1, x_2, x_3) = (1,1,1,1)$ are given to the recurrence, then the M-sequence obtained by the recurrence will be 1,1,1,1, 0,0,0,1, 0,0,1,1, 0,1,0,1, 1,1,1,0, … , and the period of the sequence is $2^4 - 1 = 15$. Taking, for example, $t = 4$ which is coprime to 15, and $w = 4$, the simple Tausworthe sequence $\{X_n\}$ with parameters (4, 1, 4) is obtained as follows:

$X_0 = x_0x_1x_2x_3 = 1111$ (= 15)

$X_1 = x_4x_5x_6x_7 = 0001$ (= 1)

$X_2 = x_8x_9x_{10}x_{11} = 0011$ (= 3)

$X_3 = x_{12}x_{13}x_{14}x_0 = 0101$ (= 5)

$X_4 = x_1x_2x_3x_4 = 1110$ (= 14)

$X_5 = x_5x_6x_7x_8 = 0010$ (= 2)

.....

The simple Tausworthe sequence obtained in this way will be, in decimal notation, 15, 1, 3, 5, 14, 2, 6, 11, 12, 4, 13, 7, 8, 9, 10, 15, 1, 3, … , and its period is $2^4 - 1 = 15$.

Suppose now that there is a multiple, say J, of simple Tausworthe sequences $\{X_n^{(j)}\}$, $j = 1, 2, ..., J$ with the same word length w. The combined Tausworthe method is a technique that generates a sequence of pseudo-random numbers $\{X_n\}$ as the bitwise exclusive logical disjunction in the binary representation of these J sequences.

$$X_n = X^{(1)}{}_n \oplus X^{(2)}{}_n \oplus \dots \oplus X^{(J)}{}_n \quad (n = 0, 1, 2, \dots)$$

The parameters and the seeds of the combined Tausworthe sequence are combinations of the parameters and the seeds of each simple Tausworthe sequence. If the periods of the J simple Tausworthe sequences are coprime, then the period of the combined Tausworthe sequence is the product of the periods of the J sequences.

NOTE 2 This method can generate sequences with good multidimensional equidistribution characteristics. The algorithm taus88_31( ) given in Annex A generates a sequence of 31-bit integers by combining three simple Tausworthe generators with parameters $(p, q, t) = (31, 13, 12)$, $(29, 2, 4)$, and $(28, 3, 17)$, respectively. The period length of the combined sequence is $(2^{31} - 1)(2^{29} - 1)(2^{28} - 1)$, i.e. about 2^{88}. Many other combinations are suggested in References [7] and [8] in the Bibliography.

5.5 Mersenne Twister method

Let X_n be a binary integer of w bits. Then, the Mersenne Twister method generates a sequence of binary integer pseudo-random numbers of w bits according to the following recurrence formula with integer constants p, q, r and a binary integer a of w bits.

$$X_{n+p} = X_{n+q} \oplus (X^{\mathrm{f}}{}_n | X^{\mathrm{l}}{}_{n+1})^{(r)}\, \mathbf{A}\,, \quad (n = 1, 2, 3, \dots)$$

where $(X^{\mathrm{f}}{}_n | X^{\mathrm{l}}{}_{n+1})^{(r)}$ represents a binary integer that is obtained by a concatenation of $X^{\mathrm{f}}{}_n$ and $X^{\mathrm{l}}{}_{n+1}$, the first $w - r$ bits of X_n and the last r bits of X_{n+1} in this order. $\mathbf{A}$ is a $w \times w$ 0-1 matrix, which is determined by a, and the product $X\mathbf{A}$ is given by the following formula.

$X >> 1$ (when the last bit of $X = 0$)

$X\mathbf{A} = (X >> 1) \oplus a$ (when the last bit of $X = 1$)

Here, X is regarded as a w dimensional 0-1 vector.

NOTE The necessary amount of memory for this computation is p words, the period becomes $2^{pw-r} - 1$, and the efficiency is better than that of the GFSR methods described previously. To improve the randomness of the first $w - r$ bits, the following series of conversions can be applied to X_n.

$$y := X_n$$

$$y := y \oplus (y >> u)$$

$$y := y \oplus [(y << s) \wedge b]$$

$$y := y \oplus [(y << t) \wedge c]$$

$$y := y \oplus (y >> l)$$

where b, c are constant bits masks to improve the randomness of the first $w - r$ bits. The parameters of this algorithm are $(p, q, r, w, a, u, s, t, l, b, c)$. The seeds are $X_2, ..., X_{q+1}$ and the first $w - r$ bits of X_1.

The final value of y is the pseudo-random number.

6 Generation of random numbers from various distributions

6.1 Introduction

Methods of generating random numbers Y from various distributions by using uniform random numbers X of integer type, are described below.

The distribution function is denoted by $F(y)$. If it is a continuous distribution, its probability density function is denoted by $f(y)$, and if it is a discrete distribution, its probability mass function is denoted by $p(y)$.

6.2 Uniform distribution

6.2.1 Standard uniform distribution

6.2.1.1 Probability density function

$$f(y) = \begin{cases} 1, & 0 \leqslant y \leqslant 1 \\ 0, & \text{otherwise} \end{cases}$$

6.2.1.2 Random variate generation method

If the maximum value of uniform random number X of integer type is $m - 1$, the following formula should be used to generate standard uniform random numbers.

$$U = \frac{X}{m}$$

EXAMPLE For any w-bit integer sequences generated by the method described in 5.2 through 5.5, $m = 2$.

NOTE 1 Because X takes on discrete values, the values of U are also discrete.

NOTE 2 The value of U never becomes 1,0. The value of U becomes 0,0 only when $X = 0$. In the case of M-sequence random numbers, any generation method may cause this phenomenon.

NOTE 3 Random numbers from the standard uniform distribution are called standard uniform random numbers, and are represented by U_1, U_2, … They are assumed to be independent of each other.

6.2.2 General uniform distribution

6.2.2.1 Probability density function

$$f(y)=\begin{cases} 1/b, & a \leqslant y \leqslant a+b \\ 0, & \text{otherwise} \end{cases}$$

where $b > 0$.

6.2.2.2 Random variate generation method

If the standard uniform random number U is generated by the method specified in 6.2.1.2, then the general uniform random number should be generated by the following formula:

$$Y = bU + a$$

6.3 Standard beta distribution

6.3.1 Probability density function

$$f(y)=\begin{cases} \dfrac{y^{c-1}(1-y)^{d-1}}{\mathrm{B}(c,d)}, & 0 \leqslant y \leqslant 1 \\ 0, & \text{otherwise} \end{cases}$$

where $\mathrm{B}(c,d) = \int_0^1 x^{c-1}(1-x)^{d-1}\mathrm{d}x$ is the beta function and the parameters c and d are greater than 0.

6.3.2 Random variate generation method by Jöhnk

If the standard uniform random numbers U_1 and U_2 are independently generated by the method specified in 6.2.1, then the standard beta random number Y should be generated by the following procedures.

If $\tilde{Y} = U_1^{1/c} + U_2^{1/d}$ is less than or equal to 1, set $Y = U_1{}^{1/c}/\tilde{Y}$; otherwise, generate two standard uniform random numbers again until the inequality is satisfied.

6.3.3 Random variate generation method by Cheng

If the standard uniform random numbers U_1 and U_2 are independently generated by the method specified in 6.2.1, then the standard beta random number Y should be generated by the following procedures.

[Set-up]

a) Let

$$q=\begin{cases} \min(c,d), & \text{if } \min(c,d) < 1 \\ \sqrt{\dfrac{2cd-(c+d)}{c+d-2}}, & \text{otherwise} \end{cases}$$

[Generation]

b) Let

$$V = \frac{1}{q}\frac{U_1}{1-U_1}, W = c\exp(V)$$

c) If

$$(c+d)\ln\left(\frac{c+d}{d+W}\right)+(c+q)V-\ln 4 \geqslant \ln(U_1^2U_2)$$

then employ

$Y=\dfrac{W}{d+W}$; and stop.

d) Generate U_1, U_2, and go to b).

$$V=\frac{1}{q}\,\frac{U_1}{1-U_1}(c+d)\ln\left(\frac{c+d}{d+W}\right)+(c+q)V-\ln 4 \geqslant \ln(U_1^2U_2)\,\frac{W}{d+W}$$

Jöhnk's method is recommended when max(c, d) $\leqslant$ 1; otherwise, Cheng's method is recommended.

NOTE General beta random variates with the support [a, $a+b$] will be obtained by a linear transformation similar to the one described in 6.2.2.2.

6.4 Triangular distribution

6.4.1 Probability density function

$$f(y)=\begin{cases}\dfrac{b-|a-y|}{b^2}, & a-b\leqslant y\leqslant a+b\\ 0, & \text{otherwise}\end{cases}$$

where $b > 0$.

6.4.2 Random variate generation method

If the standard uniform random numbers U_1 and U_2 are independently generated by the method specified in 6.2.1, then the triangular random number Y should be generated by $Y=a+b(U_1+U_2-1)$.

6.5 General exponential distribution with location and scale parameters

6.5.1 Probability density function

$$f(y)=\begin{cases}\dfrac{1}{b}\exp\{-(y-a)/b\}, & y\geqslant a\\ 0, & y<a\end{cases}$$

where a and b are the location and scale parameters of the exponential distribution, respectively.

6.5.2 Random variate generation method

If the standard uniform random number U is generated by the method specified in 6.2.1, then the general exponential random number should be generated by

$$Y=-\,b\ln(U)+a$$

6.6 Normal distribution

6.6.1 Probability density function

$$f(z) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{1}{2\sigma^2}(z-\mu)^2\right\}, \quad -\infty < z < \infty$$

where μ and σ are the mean and standard deviation of the normal distribution, respectively.

NOTE The symbol Z is used for a normal random variate.

6.6.2 The Box-Müller method

If the standard uniform random numbers U_1 and U_2 are independently generated by the method specified in 6.2.1, then two independent normal random numbers Z_1, Z_2 will be generated by the following procedures:

$$Z_1 = \mu + \sigma\sqrt{-2\ln(1-U_1)}\cos(2\pi U_2)$$

$$Z2 = \mu + \sigma\sqrt{-2\ln(1-U_1)}\sin(2\pi U_2)$$

NOTE 1 Since U_1 is not continuous, Z_1, Z_2 cannot be normally distributed in a strict sense. For example, using this procedure, the upper bound of the absolute value of the pseudo-standardized normal variates is $M = \sqrt{-2\ln(m^{-1})} = \sqrt{2\ln m}$; thus, when $m = 2^{32}$, $M \approx 6{,}660\,4$, and when $m = 2^{31} - 1$, $M \approx 6{,}555\,5$. However, since the probability that absolute values of random variables from a true standard normal distribution exceed M is only about 10^{-10}, this will rarely be a problem in practice.

NOTE 2 When generating U_1, U_2, by a linear congruential method sequentially, U_1 and U_2 are not independent of each other, so the tail of the distribution of the generated Z_1 and Z_2 can depart considerably from the true normal distribution.

6.7 Gamma distribution

6.7.1 Probability density function

$$f(y) = \begin{cases} \dfrac{1}{b\Gamma(c)}\{(y-a)/b\}^{c-1}\exp\{-(y-a)/b\}, & y \geqslant a \\ 0, & \text{otherwise} \end{cases}$$

where a, b, c are the location, scale and shape parameters of the distribution, respectively.

6.7.2 Random variate generation methods

6.7.2.1 General

Algorithms are given for three special cases depending on the shape parameter value c as follows.

6.7.2.2 Case of $c = k$ (k: integer)

Using independent uniform random numbers U_1, U_2, ... , U_k, the transformation formula

$$Y = a - b\ln\{(1-U_1)(1-U_2)\ldots(1-U_k)\}$$

should be used.

NOTE The chi-squared distribution with even degrees of freedom can be generated by this method when $a = 0$ and $b = 2$.

6.7.2.3 Case of $c = k + 1/2$ (k: integer)

Using a standard normal random number Z_0 and uniform random number U_1, U_2, ... , U_k, the transformation formula

$$Y = a + b\left[Z_0^2 / 2 - \ln\{(1-U_1)(1-U_2)...(1-U_k)\}\right]$$

should be used. In the case where $k = 0$, the logarithm term disappears.

NOTE The chi-squared distribution with odd degrees of freedom can be generated by this method when $a = 0$ and $b = 2$.

6.7.2.4 Approximate generation method when $c > 1/3$

a) Set $r = c - 1/3$, $s = \sqrt[3]{r}$, $t = r - r\ln(r)$, $p = 1/(3\sqrt{s})$ and $q = -3\sqrt{r}$.

b) Generate standard normal random number Z.

c) If $Z < q$, then go to b).

d) Set $Y = (pZ + s)^3$, $V = Z^2 / 2$, and generate U.

e) If $(Y - r)^2/Y - V \leqslant U$, then employ $Y := a + bY$ and stop.

f) Set $W = Y - r \ln(Y) - t - V$.

g) If $W \leqslant U$, then employ $Y := a + bY$ and stop.

h) If $W > -\ln(1{,}0 - U)$, then go to b).

NOTE This method is based on the Wilson-Hilferty transformation of chi-square distributions to an approximate standard normal distribution. The accuracy of this approximation depends on the parameter values of c, and the dependency is rather complicated. A very rough idea is as follows: the absolute difference between the percentage point of the approximate distribution and the exact distribution is always less than 0,2.

6.7.2.5 Exact generation method when $c > 1/2$, by Cheng

a) Set $q = c - \ln 4$ and $r = c + \sqrt{2c - 1}$.

b) Generate standard uniform random numbers U_1 and U_2.

c) Set $V = c \ln(\frac{U_1}{1-U_1}), W = c \exp(U_1), Z = U_1^2 U_2$ and $R = q + rV - W$.

d) If $R \geqslant 4{,}5Z - (1 + \ln 4{,}5)$ then employ $Y = a + bW$ and stop.

e) If $R \geqslant \ln Z$ then employ $Y = a + bW$ and stop.

f) Go to b).

$p = 1/\sqrt{2c-1},\ q = c - \ln 4 \qquad r = c + \sqrt{2c-1} \qquad q + pr\ln\left(\frac{U_1}{1-U_1}\right) - c\left(\frac{U_1}{1-U_1}\right)^p \geqslant 4{,}5\left(U_1^2 U_2\right) - (1+\ln 4{,}5)$

$$Y = a + bc\left(\frac{U_1}{1-U_1}\right)^p$$

6.8 Weibull distribution

6.8.1 Probability distribution function

$$F(y) = \begin{cases} 1 - \exp\left\{-\left(\frac{y-a}{b}\right)^c\right\}, & y \geqslant a \\ 0, & y < a \end{cases}$$

where a, b and c are the location, scale and shape parameters of the distribution, respectively.

6.8.2 Random variate generation method

If the standard uniform random numbers U are generated by the method specified in 6.2.1, then general Weibull random numbers are generated by the following formula:

$$Y = a - b\{\ln(1-U)\}^{1/c}$$

6.9 Lognormal distribution

6.9.1 Probability density function

$$f(y) = \begin{cases} \frac{1}{\sqrt{2\pi}\{(y-a)/b\}}\exp\left\{-\frac{1}{2}\left(\frac{y-a}{b}\right)^2\right\}, & y \geqslant a \\ 0, & y < a \end{cases}$$

where a and b are the location and scale parameters of the normal distribution, respectively.

6.9.2 Random variate generation method

Using standard normal random numbers Z,

$$Y = a + \exp(bZ)$$

generates lognormal random numbers.

6.10 Logistic distribution

6.10.1 Probability function

$$F(y) = \frac{1}{1 + \exp\{-(y-a)/b\}}, \quad -\infty < y < \infty$$

where a and b are the location and scale parameters of the distribution, respectively.

6.10.2 Random variate generation method

If standard uniform random numbers U are generated by the method specified in 6.2.1, then logistic random numbers are generated by

$$Y = a + b \ln\left(\frac{U}{1-U}\right)$$

6.11 Multivariate normal random variate generation

Random numbers $Y_1, Y_2, \ldots, Y_n$ from an n-dimensional normal distribution, with mean values $\mu_1, \mu_2, \ldots, \mu_n$ and variances and covariances σ_{ij} $(1 \leqslant i, j \leqslant n)$ are generated as follows using mutually independent standard normal random numbers $Z_1, \ldots, Z_n$.

$$Y_1 = \mu_1 + a_{11}Z_1$$

$$Y_2 = \mu_2 + a_{21}Z_1 + a_{22}Z_2$$

...

$$Y_n = \mu_n + a_{n1}Z_1 + a_{n2}Z_2 + \ldots + a_{nn}Z_n$$

where $a_{11}, \ldots, a_{nn}$ are constants that are calculated before start of random number generation from variances and covariances by following Cholesky factorization procedures, as given below.

NOTE σ_{ij} $(1 \leqslant i, j \leqslant n)$, σ_{ii} means the variance of Y_i.

a) Set $a_{11} = \sqrt{\sigma_{11}}$, $a_{i1} = \sigma_{i1}/a_{11}$ $(2 \leqslant i \leqslant n)$

b) For $j = 2, \ldots, n$ set

$$a_{jj} = \left(\sigma_{jj} - \sum_{k=1}^{j-1} a_{jk}^2\right)^{\frac{1}{2}}$$

and

$$a_{ij} = \left(\sigma_{ij} - \sum_{k=1}^{j-1} a_{ik}a_{jk}\right) / a_{jj} \quad (j+1 \leqslant i \leqslant n)$$

6.12 Binomial distribution

6.12.1 Probability mass function

When some event occurs with probability p at each trial, the probability that this event occurs y times in n trials is given by the following formula:

$$p(y) = \binom{n}{y} p^y (1-p)^{n-y}, \; y = 0, 1, \ldots, n$$

where $0 < p < 1$.

6.12.2 Random variate generation methods

6.12.2.1 General

The following methods should be used for generating random numbers Y from this distribution.

6.12.2.2 Direct method

Generate n standard uniform random numbers U, and let Y be the number of these values of U that are less than p.

6.12.2.3 Inverse function method

Compute the distribution function as follows:

$$F(y) = \sum_{k=0}^{y} \binom{n}{k} p^k (1-p)^{n-k}, \; y = 0, 1, \ldots, n$$

Whenever a random number is required, generate a standard uniform random number U, and let Y be the minimum y that satisfies $U \leqslant F(y)$.

6.12.2.4 Alias method

First, $n + 1$ parameters $v_0, v_1, \ldots, v_n$ and $n + 1$ other parameters $a_0, a_1, \ldots, a_n$, are calculated by the following procedures.

a) $v_y = (n + 1)\, p(y), y = 0, 1, \ldots, n.$

b) Let G be the set of suffices y that satisfies $v_y \geqslant 1$ and S be the set of suffices y that satisfies $v_y < 1$.

c) While S is not empty, repeat the following operations from 1) to 4).

   1) Select any element i from G and any element j from S.

   2) Set $a_j = i$ and $v_i = v_i - (1 - v_j)$.

   3) If $v_i < 1$, delete element i from G and move it to S.

   4) Delete element j from S.

If the preparations mentioned above are completed, a binomial random number Y will be obtained by performing the following operations d) to f).

d) Generate a standard uniform random number U, and set $V = (n + 1)U$.

e) Set $k =$ (integer part of V) and $u = V - k$.

f) If $u \leqslant v_k$, set $Y = k$; otherwise, set $Y = a_k$.

6.13 Poisson distribution

6.13.1 Probability mass function

The probability mass function of a Poisson distribution with mean μ is defined as follows.

$$p(y) = \exp(-\mu)\frac{\mu^y}{y!},\ y = 0,\ 1,\ 2,\ \dots,$$

where $\mu > 0$.

6.13.2 Method of using a relationship with an exponential distribution

Generate standard uniform random numbers U_1, U_2, ..., and let Y be the maximum n that satisfies the following inequality:

$$-\ln\{(1-U_1)(1-U_2)\dots(1-U_n)\} < \mu$$

6.13.3 Alias method

First, select a constant n for which the probability that $Y > n$ is negligibly small, for example, the integer part of $\mu + 6\sqrt{\mu}$ could be specified to be n. Then use the procedure 6.12.2.4 alias method of the binomial distribution; however, this time the probability mass function of the Poisson distribution shall be used for $p(y)$.

NOTE This method is efficient when μ is of medium size (about 10 to 100).

6.14 Discrete uniform distribution

For generating discrete uniform random variates from M to N, a binary r bit random number sequence generated by the method standardized in 5.1 is converted by the following procedures, where $N - M + 1$ is assumed to be not greater than 2^r.

a) Find the natural number k that satisfies the following inequality:

$$2^{k-1} + 1 \leqslant N - M + 1 \leqslant 2^k$$

NOTE 1 Such k is the minimum natural number that is equal to or greater than $\log_2 (N - M + 1)$.

EXAMPLE 1 When $N - M + 1 = 100$, $k = 7$ because $2^6 + 1 = 65 \leqslant 100 \leqslant 2^7 = 128$.

b) Add 1 to the binary integer that is constructed from the first k bits of a random number, and convert the result to a decimal number.

NOTE 2 A k bit binary number $Z_1Z_2Z_3Z_4\dots Z_k$ converts to a decimal number $2^{k-1}Z_1 + 2^{k-2}Z_2 + 2^{k-3}Z_3 + 2^{k-4}Z_4 + \dots + Z_k$.

EXAMPLE 2 When the upper 7 bits are 1 011 001, the corresponding decimal number is $64 + 16 + 8 + 1 = 89$, and the decimal random number becomes 89.

c) The required decimal random number is the converted decimal number plus $M - 1$ by skipping numbers greater than N.

NOTE 3 When $N - M + 1$ is more than 2^r the required decimal random number can be obtained by concatenating two or more binary random numbers to get one binary random number.

NOTE 4 When using the linear congruential method for generating pseudo-random numbers, k shall not be specified to be equal to r.

Further, when $N - M + 1$ is a decimal k digit natural number, and k is not too large, say k is less than 20, the method given in 5.2 can be used. The procedure is as follows.

d) Generate a decimal random number sequence of k digits by using procedure 5.2.

e) From the random number sequence which is generated by d) above, remove the numbers greater than N. The sequence thus obtained is the required decimal random number sequence.

Annex A
(informative)

Table of physical random numbers

A.1 Table of random numbers

Physically generated random numbers have no functional relationship like pseudo-random numbers, and no periodicity. Table A.1 shows a physically generated random number sequence obtained as measured values of a property of a random physical system.

Table A.1 — Physical random number table

1	93	90	60	02	17	25	89	42	27	41	64	45	08	02	70	42	49	41	55	98
2	34	19	39	65	54	32	14	02	06	84	43	65	97	97	65	05	40	55	65	06
3	27	88	28	07	16	05	18	96	81	69	53	34	79	84	83	44	07	12	00	38
4	95	16	61	89	77	47	14	14	40	87	12	40	15	18	54	89	72	88	59	67
5	50	45	95	10	48	25	29	74	63	48	44	06	18	67	19	90	52	44	05	85
6	11	72	79	70	41	08	85	77	03	32	46	28	83	22	48	61	93	19	98	60
7	19	31	85	29	48	89	59	53	99	46	72	29	49	06	58	65	69	06	87	09
8	14	58	90	27	73	67	17	08	43	78	71	32	21	97	02	25	27	22	81	74
9	28	04	62	77	82	73	00	73	83	17	27	79	37	13	76	29	90	07	36	47
10	37	43	04	36	86	72	63	43	21	06	10	35	13	61	01	98	23	67	45	21
11	74	47	22	71	36	15	67	41	77	67	40	00	67	24	00	08	98	27	98	56
12	48	85	81	89	45	27	98	41	77	78	24	26	98	03	14	25	73	84	48	28
13	55	81	09	70	17	78	18	54	62	06	50	64	90	30	15	78	60	63	54	56
14	22	18	73	19	32	54	05	18	36	45	87	23	42	43	91	63	50	95	69	09
15	78	29	64	22	97	95	94	54	64	28	34	34	88	98	14	21	38	45	37	87
16	97	51	38	62	95	83	45	12	72	28	70	23	67	04	28	55	20	20	96	57
17	42	91	81	16	52	44	71	99	68	55	16	32	83	27	03	44	93	81	69	58
18	07	84	27	76	18	24	95	78	67	33	45	68	38	56	64	51	10	79	15	46
19	60	31	55	42	68	53	27	82	67	68	73	09	98	45	72	02	87	79	32	84
20	47	10	36	20	10	48	09	72	35	94	12	94	78	29	14	80	77	27	05	67
21	73	63	78	70	96	12	40	36	80	49	23	29	26	69	01	13	39	71	33	17
22	70	65	19	86	11	30	16	23	21	55	04	72	30	01	22	53	24	13	40	63
23	86	37	79	75	97	29	19	00	30	01	22	89	11	84	55	08	40	91	26	61
24	28	00	93	29	59	54	71	77	75	24	10	65	69	15	66	90	47	90	48	80
25	40	74	69	14	01	78	36	13	06	30	79	04	03	28	87	59	85	93	25	73

A.2 Method of physical random number generation

The method by which the physical random numbers in Table A.1 were generated is described below. The source of the numbers is the noise generated by a diode. In a diode, the noise signal is large because, by amplification using the electron avalanche effect, stabilized noise is easily obtained. For this reason, it is used very often as a noise source. For the element, NC2401[1] of Noisecom in the United States was used. This element has a noise source and an amplifier built-in, and its band width is 1 GHz, while its amplitude is 160 mVrms.

The methods of digitalizing the noise signal are

a) analogue/digital conversion,

b) regarding noise as a pulse sequence, by counting pulses per unit time,

c) regarding noise as a pulse sequence, by measuring pulse interval.

For example, consider the use of a DAS-4102 converter[2] produced by the Keithley Instruments, Inc. for analogue/digital conversion. This equipment has a resolution of 8 bits with a sampling period of 64 MHz maximum. Data was sampled at 1 MHz to produce the attached table. Measuring was done with resolution ability 3,91 mV/digit and only the lowest bit was used as a random number source.

Because the analogue/digital conversion equipment has errors that are peculiar to the equipment, the frequency distribution of values after conversion is not uniform. To obtain a more uniform distribution, 2 bits were generated from the same random number source, and

(0,1)	→	Random number (Rn) = 0
(1,0)	→	Random number (Rn) = 1
(0,0), (1,1)	→	abandoned

Random numbers in Table A.1 were generated according to the above scheme. If the probabilities of (0,1) and (1,0) are equal to each other, the random number is uniformly distributed. Because the intervals between successive measurements are as short as 1 ms, the characteristics of the equipment would scarcely change in this time interval. Therefore, (0,1) and (1,0) are considered to conform to the same probability distribution. An alternative method of correcting is by formerly measuring the probability distribution of the characteristics, but, because this distribution varies from equipment to equipment, this method was not employed. Further, for safety, 32 bits were gathered in one group, and exclusive or was done with pseudo-random numbers using the Mersenne Twister (routine name genrand) which is described in 5.5. The Mersenne Twister was initialized by the routine init_genrand(s), $s = 19660809$. If the original random number sequences are required, they can be regenerated by operating exclusive or again using the Mersenne Twister.

Table A.1 is a decimal random number sequence generated by the above-mentioned method, taking the upper 4 bits of the 32-bit random number sequence. If the value of this is not less than 0 and not more than 9, the value is employed as the random number value; however, if the value of this is 10 or more, it is abandoned and the next random number is generated.

1) NC2401 is the trade name of a product supplied by Noisecom. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named.

2) DAS-4102 is the trade name of a product supplied by Keithley Instruments, Inc. This information is given for the convenience of users of this document and does not constitute an endorsement by ISO of the product named.

Annex B
(informative)

Algorithm for pseudo-random number generation

B.1 Program code for the trinomial GFSR method

The following C program given below, which is in accordance with ISO/IEC 9899, is an example with parameters $(p, q, w) = (1\,279, 418, 32)$ and period $(2^{1\,279} - 1)$. When the function gfsr( ) is called, it generates an integer between 0 and $(2^{32} - 1)$ inclusive. When the function gfsr_31( ) is called, it generates an integer between 0 and $(2^{31} - 1)$ inclusive. Before calling the functions gfsr( ) and gfsr_31( ), initialization is necessary by calling init_gfsr(s) once.The initialization function init_gfsr(s) executes initialization under the condition that an unsigned 32-bit integer [integer between 0 and $(2^{32} - 1)$] is used as the seed. The generated sequence can be used to provide 39 independent series, each of which has negligible auto-correlation, is 39-distributed (uniformly distributed in a 39-dimensional hyper-cube) with 32-bit precision, and its auto-correlation function assumes values close to zero up to phase shift $2^{1\,274}$.

To obtain different pseudo-random number series, change the seed s given to init_gfsr(s). Only the constants p, q, w in the program may be changed. The value of w will be a power of 2 within the word length of the machine. The value of w will generally be 32 or 64, according to the machine. For example, if the word length of the machine is 64, the constant w in the program is set to 64, and then gfsr( ) generates integers between 0 and $(2^{64} - 1)$ inclusive, while gfsr_31( ) generates integers between 0 and $(2^{63} - 1)$ inclusive.

In this program, the length of type “unsigned long” is presumed to be not less than 32 bits.

```
/***************************************************
 C code : Trinomial GFSR
***************************************************/


#define P 1279

#define Q 418

#define W 32 /* W should be a power of 2 */


static unsigned long state [P] ;

static int state_i ;


void init_gfsr (unsigned long s)

{

 int i, j, k;

 static unsigned long x [P] ;


 s &= 0xffffffffUL;


 for (i=0 ; i<P ; i++) {

   x [i] = s>>31 ;
```

```
    s = 1664525UL * s + 1UL ;
    s &= 0xffffffffUL ;
  }

  for (k=0, i=0 ; i<P ; i++) {
   state [i] = 0UL ;
   for (j=0 ; j<W ; j++) {
    state [i] <<= 1 ;
    state [i] |= x [k] ;
    x [k] ^= x [ (k+Q) %P] ;
    k++;
    if (k==P) k = 0 ;
   }
  }

  state_i = 0 ;
}

unsigned long gfsr (void)
{
int i ;
unsigned long *p0, *p1 ;

  if (state_i >= P) {
   state_i = 0 ;
   p0 = state ;
   p1 = state + Q ;
   for (i=0 ; i<(P-Q) ; i++)
    *p0++ ^= *p1++ ;
   p1 = state ;
   for ( ; i<P ; i++)
    *p0++ ^= *p1++ ;
  }

return state [state_i++] ;
}

/* W-1 bit integer */
long gfsr_31 (void)
{
 return (long) (gfsr( ) >>1) ;
}
```

NOTE The corresponding Full Basic code of the trinomial GFSR method is shown for information as follows.

```
REM /*********************************************

REM  BASIC code : Trinomial GFSR

REM *********************************************/


OPTION BASE 0


REM
/******************************************************************************/

DECLARE NUMERIC P

LET  P=1279                            !#define P 1279

DECLARE NUMERIC Q

LET  Q=418                             !#define Q 418

DECLARE NUMERIC W

LET  W=32                              !#define W 32 /* W should be power OF 2 */


DIM state(P)                           !static unsigned long state[P];

DECLARE NUMERIC state_i                !static INT state_i;


REM
/******************************************************************************/

FUNCTION init_gfsr(s)                  !void init_gfsr(unsigned long s){

   DECLARE NUMERIC i,j,k               !    int i, j, k;

   DIM x(P)                            !    static unsigned long x[P];

   LET  s = And32(s , MskF_f)          !    s &= 0xffffffffUL;

   FOR i = 0 TO P -1                   !    for (i=0; i<P; i++) {

      LET  x(i) = SR32U(s , 31)        !        x[i] = s>>31;

      LET  s = Mul32U(1664525 , s) + 1        !        s = 1664525UL * s + 1UL;
      LET  s = And32(s, MskF_f)               !       s &= 0xffffffffUL;
   NEXT I                                     !    }
   LET  k=0
   FOR i = 0 TO P -1                          !    for (k=0,i=0; i<P; i++) {
      LET  state(i) = 0                       !       state[i] = 0UL;
      FOR j=0 TO W-1                          !       for (j=0; j<W; j++) {
         LET  state(i) = SL32U(state(i) , 1)  !           state[i] <<= 1;
         LET  state(i) = Or32(state(i) , x(k))!           state[i] |= x[k];
         LET  x(k) = Xor32(x(k) , x(REMAINDER(k + Q , P)))
         !                                    !           x[k] ^= x[(k+Q)%P];
         LET  k = k + 1                       !           k++;
         IF k = P THEN LET  k = 0             !           if (k==P) k = 0;
      NEXT j                                  !       }
   NEXT I                                     !    }
   LET  state_i = 0                           !    state_i = 0;
END FUNCTION                                  !}


REM
/*********************************************************************************/

FUNCTION gfsr                                 !unsigned long gfsr(void){
   DECLARE NUMERIC I                          !    int i;
   DECLARE NUMERIC p0, p1                     !    unsigned long *p0, *p1;
   IF state_i >= P THEN                       !    if (state_i >= P) {
      LET  state_i = 0                        !       state_i = 0;
      LET  p0 = 0                             !       p0 = state;
      LET  p1 = Q                             !       p1 = state + Q;
      FOR i=0 TO P-Q-1                        !       for (i=0; i<(P-Q); i++)
         LET  state(p0) = Xor32(state(p0) , state(p1))
         LET  p0 = p0 + 1
         LET  p1 = P1 + 1               !            *p0++ ^= *p1++;
      NEXT i
      LET  p1 = 0                         !        p1 = state;
      FOR i=i TO P-1                      !        for (; i<P; i++)
         LET  state(p0) = Xor32(state(p0) , state(p1))
         LET  p0 = p0 + 1
         LET  p1 = P1 + 1                 !            *p0++ ^= *p1++;
      NEXT i
   END IF                                 !    }
   LET  gfsr = state(state_i)
   LET  state_i = state_i + 1             !    return state[state_i++];
END FUNCTION                              !}


REM
/********************************************************************************/
REM /* W-1 bit integer */
FUNCTION gfsr_31                          !long gfsr_31(void){
   LET  gfsr_31 = SR32U(gfsr , 1)         !    return (long)(gfsr()>>1);
END FUNCTION                              !}
```

B.2 Program code for the pentanomial GFSR method

In this program, the parameters are (521, 86, 197, 447, 32), and the period is $2^{521} - 1$. When the function gfsr5( ) is called, it generates an integer between zero and ($2^{32} - 1$) inclusive. When the function gfsr5_31( ) is called, it generates an integer between zero and ($2^{31} - 1$) inclusive. The initialization routine init_gfsr5(s) initializes under the condition that the seed is an unsigned 32-bit integer [integer between 0 and ($2^{32} - 1$)]. Before calling gfsr5( ) and gfsr5_31( ), init_gfsr5(s) is executed once to initialize. The generated sequence is 16-distributed (uniformly distributed in a 16-dimensional hyper-cube) with 32-bit precision, and its auto-correlation function assumes almost zero value up to the phase shift 2^{516}.

If an independent batch of random numbers is needed for each one of multiple replications of a simulation, the initialization function init_gfsr5(s) should be called only once before the start of the simulation. After each replication, the contents of the table x[P] of length P and the value of the variable state_i should be saved, and used as the initial values for the next replication.

If another sequence with different period is required, a set of values p, q_1, q_2 and q_3 shall be selected from Table 1.

```
/**************************************************
 C code : Pentanominal GFSR
**************************************************/
```

```
#define P 521
/* Q1 < Q2 < Q3 */
#define Q1 86
#define Q2 197
#define Q3 447
#define W 32 /* W should be a power of 2 */

static unsigned long state [P] ;\
static int state_i ;

void init_gfsr5 (unsigned long s)
{
 int i, j, k ;
 static unsigned long x [P] ;
 s &= 0xffffffffUL ;

 for (i=0 ; i<P ; i++) {
  x [i] = s>>31 ;
  s = 1664525UL * s + 1UL ;
  s &= 0xffffffffUL ;
 }

 for (k=0, i=0 ; i<P ; i++) {
  state [i] = 0UL ;
  for (j=0 ; j<W ; j++) {
   state [i] <<= 1 ;
   state [i] |= x [k] ;
   x [k] ^=x [ (k+Q1) %P] ^x [ (k+Q2) %P] ^x [ (k+Q3) %P] ;
   k++;
   if (k==P) k = 0 ;
  }
 }

 state_i = 0 ;
}

unsigned long gfsr5 (void)
{
 int i ;
 unsigned long *p0, *p1, *p2, *p3 ;

 if (state_i >= P) {
  state_i = 0 ;
  p0 = state ;
  p1 = state + Q1 ;
  p2 = state + Q2 ;
  p3 = state + Q3 ;

 for (i=0 ; i<(P-Q3) ; i++)
  *p0++ ^= *p1++ ^ *p2++ ^ *p3++;
 p3 = state ;
 for ( ; i<(P-Q2) ; i++)
  *p0++ ^= *p1++ ^ *p2++ ^*p3++;
 p2 = state;
 for ( ; i<(P-Q1) ; i++)
  *p0++ ^= *p1++ ^ *p2++ ^*p3++;
 p1 = state;
 for ( ; i<P ; i++)
  *p0++ ^= *p1++ ^*p2++ ^*p3++;
 }

 return state [state_i++] ;
}

/* W-1 bit integer */
long gfsr5_31 (void)
{
 return (long) (gfsr5( ) >>1);
  }
```

NOTE The corresponding Full Basic code of the pentanomial GFSR method is shown for information as follows.

```
REM /*********************************************

REM BASIC code : Pentanomial GFSR

REM *********************************************/


OPTION BASE 0
```

```
REM
/*******************************************************************************/

DECLARE NUMERIC P

LET  P = 521                              !#define P 512

REM /* Q1 < Q2 < Q3 */

DECLARE NUMERIC Q1

LET  Q1 =  86                             !#define Q1 86

DECLARE NUMERIC Q2

LET  Q2 = 197                             !#define Q2 197

DECLARE NUMERIC Q3

LET  Q3 = 447                             !#define Q3 447

DECLARE NUMERIC W

LET  W  =  32                             !#define W 32 /* W should be power of
2 */


DIM state(P)                              !static unsigned long state[P];

DECLARE NUMERIC state_i                   !static int state_i;


REM
/*******************************************************************************/

FUNCTION init_gfsr5(s)                    !void init_gfsr5(unsigned long s)   {

   DECLARE NUMERIC i, j, k                !    int i, j, k;

   DIM x(P)                               !    static unsigned long x[P];


   LET  s = And32(s , MskF_f)             !    s &= 0xffffffffUL;


   FOR i=0 TO P-1                         !    for (i=0; i<P; i++) {

      LET  x(i) = SR32U(s , 31)           !        x[i] = s>>31;

      LET  s = Mul32U(1664525 , s) + 1    !        s = 1664525UL * s + 1UL;

      LET  s = And32(s , MskF_f)          !        s &= 0xffffffffUL;

   NEXT I                                     !    }
   LET  k=0

   FOR i=0 TO P-1                                   !    for (k=0,i=0; i<P; i++) {

      LET  state(i) = 0                             !        state[i] = 0UL;

      FOR j=0 TO W-1                                !        for (j=0; j<W; j++) {

         LET  state(i) = SL32U(state(i) , 1)        !            state[i] <<= 1;

         LET  state(i) = Or32(state(i) , x(k))!            state[i] |= x[k];

         LET   x(k)  =  Xor32(Xor32(Xor32(x(k)  ,  x(REMAINDER(k  +  Q1 ,  P)))  ,
x(REMAINDER(k + Q2 , P))) , x(REMAINDER(k + Q3 , P)))

         !                                          !             x[k] ^= x[(k+Q1)%P] ^
x[(k+Q2)%P] ^ x[(k+Q3)%P];

         LET  k = k + 1                             !            k++;

         IF k = P THEN LET  K = 0                   !            if (k==P) k = 0;

      NEXT j                                        !        }

   NEXT I                                           !    }


   LET  state_i = 0                                 !    state_i = 0;

END FUNCTION                                        !}


REM
/******************************************************************************/

FUNCTION gfsr5                                      !unsigned long gfsr5(void)    {

   DECLARE NUMERIC I                                !    int i;

   DECLARE NUMERIC p0, p1, p2, p3                   !     unsigned long  *p0, *p1, *p2,
*p3;


   IF state_i >= P THEN                             !    if (state_i >= P) {

      LET  state_i = 0                              !        state_i = 0;

      LET  p0 = 0                                   !        p0 = state;

      LET  p1 = Q1                                  !        p1 = state + Q1;

      LET  p2 = Q2                                  !        p2 = state + Q2;

      LET  p3 = Q3                               !        p3 = state + Q3;
      FOR i=0 TO P-Q3-1                    !        FOR (i=0; i<(P-Q3); i++)

         LET  state(p0) = Xor32(Xor32(Xor32(state(p0) , state(p1)) , state(p2)) ,
state(p3))

         LET  p0 = p0 + 1

         LET  p1 = p1 + 1

         LET  p2 = p2 + 1

         LET  p3 = p3 + 1                  !            *p0++ ^= *p1++ ^ *p2++ ^
*p3++;

      NEXT i

      LET  p3 = 0                          !        p3 = state;

      FOR i=i TO P-Q2-1                    !        for (; i<(P-Q2); i++)

         LET  state(p0) = Xor32(Xor32(Xor32(state(p0) , state(p1)) , state(p2)) ,
state(p3))

         LET  p0 = p0 + 1

         LET  p1 = p1 + 1

         LET  p2 = p2 + 1

         LET  p3 = p3 + 1                  !            *p0++ ^= *p1++ ^ *p2++ ^
*p3++;

      NEXT i

      LET  p2 = 0                          !        p2 = state;

      FOR i=i TO P-Q1-1                    !        for (; i<(P-Q1); i++)

         LET  state(p0) = Xor32(Xor32(Xor32(state(p0) , state(p1)) , state(p2)) ,
state(p3))

         LET  p0 = p0 + 1

         LET  p1 = p1 + 1

         LET  p2 = p2 + 1

         LET  p3 = p3 + 1                  !            *p0++ ^= *p1++ ^ *p2++ ^
*p3++;

      NEXT i

      LET  p1 = 0                          !        p1 = state;

      FOR i=i TO P-1                       !        for (; i<P; i++)
```

```
         LET  state(p0) = Xor32(Xor32(Xor32(state(p0) , state(p1)) , state(p2)) ,
state(p3))

         LET  p0 = p0 + 1

         LET  p1 = p1 + 1

         LET  p2 = p2 + 1

         LET  p3 = p3 + 1               !              *p0++ ^= *p1++ ^ *p2++ ^
*p3++;

      NEXT i

   END IF                               !    }


   LET  gfsr5 = state(state_i)

   LET  state_i = state_i + 1           !    return state[state_i++];

END FUNCTION                            !}


REM
/********************************************************************************/

REM /* W-1 bit integer */

FUNCTION gfsr5_31                       !long gfsr5_31(void)    {

   LET  gfsr5_31 = SR32U(gfsr5 , 1)     !    return (long)(gfsr5()>>1);

END FUNCTION                            !}
```

B.3 Program code for the combined Tausworthe method

Following is a C Language implementation of the combined Tausworthe method, which generates integers between zero and ($2^{31} - 1$) inclusive by combining three Tausworthe sequences of parameters (31, 13, 12), (29, 2, 4) and (28, 3, 17).

The initialization function init_taus88(s) initializes under the condition that the seed s is an unsigned 32-bit integer [integer between 0 and ($2^{32} - 1$) inclusive]. To obtain a different pseudo-random number sequence, change the seed s. Before calling taus88_31( ), init_taus88(s) executes once to initialize. The initialization can be done without using init_taus88(s) by directly assigning suitable values into s_1, s_2 and s_3. However, when initializing, the following three conditions must be satisfied:

— at least one of the upper 31 bits of s_1 is one;

— at least one of the upper 29 bits of s_2 is one;

— at least one of the upper 28 bits of s_3 is one.

Because the lowest 1 bit of s_1, the lowest 3 bits of s_2 and the lowest 4 bits of s_3 are ignored, the generated random number sequence is unaffected by the changes to those bits.

In this program, the length of type “unsigned long” is presumed as 32 bits.

```
/**************************************************

 C code : Combined Tausworthe

**************************************************/


static unsigned long s1, s2, s3, b ;


void init_taus88 (unsigned long s)

{

 int i ;

 unsigned long x [3] ;


 i=0 ;

 while (i<3) {

  if (s & 0xfffffff0UL) {

   x [i] = s ;

   i++;

  }

  s = 1664525UL * s + 1UL;

 }

 s1 = x [0] ; s2 = x [1] ; s3 = x [2] ;

}

/* 31 bit integer */


long taus88_31 (void)

{

 b = (((s1 << 13) ^ s1) >> 19) ;

 s1 = (((s1 & 4294967294UL) << 12) ^b) ;

 b = (((s2 << 2) ^ s2) >> 25);

 s2 = (((s2 & 4294967288UL) << 4) ^b) ;
 b = (((s3 << 3) ^ s3) >> 11) ;

 s3 = (((s3 & 4294967280UL) <<17) ^b) ;


 return (long) ((s1 ^ s2 ^ s3) >>1) ;

}
```

In this program, the code

```
 b = (((s1 << 13) ^ s1) >> 19) ;
s1 = (((s1 & 4294967294UL) << 12) ^b) ;
```

generates a number in the Tausworthe sequence with parameters (31, 13, 12) in s_1, and the codes

```
  b = (((s2 << 2) ^ s2) >> 25) ;
 s2 = (((s2 & 4294967288UL) << 4) ^b) ;
```

and

```
  b = (((s3 << 3) ^s3) >> 11) ;
 s3 = (((s3 & 4294967280UL) << 17) ^b) ;
```

generate numbers in the Tausworthe sequence with parameters (29, 2, 4) and (28, 3, 17) correspond to s_2 and s_3, respectively. These three binary integers are combined bit by bitwise exclusive-or operations, and a 31-bit pseudo-random number sequence is generated.

Selection of the three parameters (p, q, t) is made to give the best multi-dimensional equidistribution of the pseudo-random number sequence after the combination. These values of parameters shall not be changed. To obtain a different pseudo-random number sequence, change the seed.

If an independent batch of random numbers are needed for each one of multiple replications of a simulation, the initialization function init_taus88(s) should be called only once before the start of the simulation. After each replication, the values of s_1, s_2, and s_3 should be saved, and given to the variables s_1, s_2, and s_3, respectively, as the initial values for the next replication.

NOTE The corresponding Full Basic code of the Combined Tausworthe method is shown for information as follows.

```
REM /****************************************************

REM  BASIC code : Combined Tausworthe

REM ****************************************************/



OPTION BASE 0
```

```
REM
/*****************************************************************************/

FUNCTION init_taus88(s)                           !void  init_taus88(unsigned  long
s) {

   DECLARE NUMERIC I                              !    int i;

   DIM x(3)                                       !    unsigned long x[3];

   FOR i = 0 TO 2                                 !    i=0;    while (i<3) {

      IF And32(s , MskF_0) <> 0 THEN              !         if  (s & 0xfffffff0UL)
{

         LET  x(i) = s                            !         x[i] = s;      i++;

      END IF                                      !    }

      LET  s = Mul32U(1664525, s) + 1             !          s  =  1664525UL  *  s  +
1UL;

   NEXT I                                         !    }

   LET  s1 = x(0)

   LET  s2 = x(1)

   LET  s3 = x(2)                                 !    s1 = x[0]; s2 = x[1]; s3 =
x[2];

END FUNCTION                                      !}


REM
/*****************************************************************************/

FUNCTION taus88_31                                    !long      taus88_int(void)
{

   REM /***** 31 bit integer *****/

   LET  b = SR32U(Xor32(SL32U(s1, 13), s1), 19        !      b  =  (((s1  <<  13)  ^
s1) >> 19);

   LET  s1 = Xor32(SL32U(And32(s1 , MskF_e), 12), b)  !          s1    =    (((s1    &
4294967294) << 12) ^ b);

   LET  b = SR32U(Xor32(SL32U(s2, 2), s2), 25)        !     b = (((s2 << 2) ^ s2)
>> 25);

   LET  s2 = Xor32(SL32U(And32(s2 , MskF_8), 4),  b)  !          s2    =    (((s2    &
4294967288) << 4)  ^ b);

   LET  b = SR32U(Xor32(SL32U(s3, 3),  s3), 11        !     b = (((s3 << 3)  ^ s3)
>> 11);
```

```
   LET  s3 = Xor32(SL32U(And32(s3 , MskF_0), 17), b)    !         s3   =   (((s3   &
4294967280) << 17) ^ b);

   LET  taus88_31 = SR32U(Xor32(Xor32(s1, s2), s3), 1)  !     return  (long)((s1  ^
s2 ^ s3) >> 1);

   !                                                    !'}

END FUNCTION
```

B.4 Program code for the Mersenne Twister method

The following program is a C language implementation of the Mersenne Twister method. The function genrand( ) of this code generates unsigned integer pseudo-random numbers of 32 bits whose range is between 0 and ($2^{32} - 1$) inclusive. The function genrand_31( ) generates unsigned integer pseudo-random numbers of 31 bits whose range is between 0 and ($2^{31} - 1$) inclusive. The function init_genrand(s) initializes the seed as an unsigned 32-bit integer [integer between 0 and ($2^{32} - 1$) inclusive]. Before calling genrand( ) or genrand_31( ), initialization shall be done by executing init_genrand(*s*) once. Different seeds s lead to different sequences. The parameters in this program should not be changed.

If independent batch of random numbers are needed for each one of multiple replications of a simulation, the initialization function init_genrand(*s*) should be called only once before the start of the simulation. After each replication, contents of the table mt[P] of length P and the value of the variable mti should be saved, and used as the initial values for the next replication.

EXAMPLE This is an example using $p = 624$ words with parameters (624, 397, 31, 32, 0x9908b0df, 11, 7, 15, 18, 0x9d2c5680, 0xefc60000). Here, 10-digit numbers starting with 0x are unsigned 32-bit constants represented in hexadecimal. The period is $2^{19\,937} - 1$ and this is distributed uniformly in 623 dimensional hyper-cube with 32 bits precision; moreover, the sequence is equidistributed in 3 115 dimensions with 6 bits precision.

In this program, the length of type “unsigned long” is presumed to be not less than 32 bits.

```
*************************************************

C code : Mersenne Twister

**************************************************/



/* Period parameters */

#define P 624

#define Q 397

#define MATRIX_A 0x9908b0dfUL /* constant vector a */

#define UPPER_MASK 0x80000000UL /* most significant w-r bits */

#define LOWER_MASK 0x7fffffffUL /* least significant r bits */


static unsigned long mt [P] ; /* the array for the state vector */
static int mti=P+1 ; /* mti==P+1 means mt [P] is not initialized */


/* initializes mt [P] with a seed */

void init_genrand (unsigned long s)

{

 mt [0] = s & 0xffffffffUL ;

 for (mti=1 ; mti<P ; mti++) {

  mt [mti] = (1664525UL * mt [mti-1] + 1UL) ;

  mt [mti] &= 0xffffffffUL ;

 }

}


/* generates a random number on [0, 0xffffffff] -interval */

unsigned long genrand (void)

{

 unsigned long y ;

 static unsigned long mag01 [2] = {0x0UL, MATRIX_A} ;

 /* mag01 [x] = x * MATRIX_A for x=0, i */


 if (mti >=P) { /* generate P words at one time */

  int kk ;


 if (mti == P+1) /* if init _genrand ( ) has not been called, */

  init_genrand (5489UL) ; /* a default initial seed is used */


 for (kk=0 ; kk<P-Q ; kk++) {

  y = (mt [kk] &UPPER_MASK) | (mt [kk+1] &LOWER_MASK) ;

  mt [kk] = mt [kk+Q] ^ (y >> 1) ^ mag01 [y & 0x1UL] ;

 }

 for ( ; kk<P-1 ; kk++) {
  y = (mt [kk] &UPPER_MASK) | (mt [kk+1] &LOWER_MASK) ;
  mt [kk] = mt [kk+ (Q-P) ] ^ (y >> 1) ^ mag01 [y & 0x1UL] ;
 }
 y = (mt [P-1] &UPPER_MASK) | (mt [0] &LOWER_MASK) ;
 mt [P-1] = mt [Q-1] ^ (y >> 1) ^ mag01 [y & 0x1UL] ;

 mti = 0 ;
 }

 y = mt [mti++] ;
 /* Tempering */
 y ^= (y >> 11) ;
 y ^= (y << 7) & 0x9d2c5680UL ;
 y ^= (y << 15) & 0xefc60000UL ;
 y ^= (y >> 18) ;

 return y ;
}

/* generates a random number on [0, 0x7fffffff] -interval */
long genrand_31 (void)
{
 return (long) (genrand( ) >>1) ;
}
```

NOTE The corresponding Full Basic code of the Mersenne Twister method is shown for information as follows.

```
REM /****************************************************/
REM  Mersenne Twister
REM /****************************************************/
```

```
OPTION BASE 0


REM
/*************************************************************************************/

REM /* Period parameters */

DECLARE NUMERIC P

LET  P = 624                              !#define P 624

DECLARE NUMERIC Q

LET  Q = 397                              !#define Q 397

DECLARE NUMERIC MATRIX_A

LET  MATRIX_A = BVAL("9908b0df" , 16)     !#define  MATRIX_A  0x9908b0dfUL     /*
constant vector a */

DECLARE NUMERIC UPPER_MASK

LET  UPPER_MASK = BVAL("80000000" , 16)   !#define  UPPER_MASK  0x80000000UL  /*
most significant w-r bits */

DECLARE NUMERIC LOWER_MASK

LET  LOWER_MASK = BVAL("7fffffff" , 16)   !#define  LOWER_MASK  0x7fffffffUL  /*
least significant r bits */



DIM mt(P)                                 !static  unsigned  long  mt[P];  /*  the
array for the state vector  */

DECLARE NUMERIC mti

LET  mti = P + 1                          !static int mti=P+1; /* mti==P+1 means
mt[P] is not initialized */



REM
/*************************************************************************************/

REM /* initializes mt[P] with a seed */

FUNCTION init_genrand(s)                  !void init_genrand(unsigned long s)  {

   LET  mt(0) = And32(s , MskF_f)         !    mt[0]= s & 0xffffffffUL;

   FOR mti = 1 TO P - 1                   !    for (mti=1; mti<P; mti++) {

      LET  mt(mti) = Mul32U(1664525 , mt(mti-1)) + 1

      !                                    !           mt[mti]  =  (1664525UL  *
mt[mti-1] + 1UL);
```

```
      LET  mt(mti) = And32(mt(mti) , MskF_f) !        mt[mti] &= 0xffffffffUL;
   NEXT mti                                  !    }
END FUNCTION                                 !}


REM
/******************************************************************************/
REM /* generates a random number ON [0,0xffffffff]-interval */
FUNCTION genrand                             !unsigned long genrand(void){
   DECLARE NUMERIC y                         !    unsigned long y;
   DIM  mag01(2)
   LET  mag01(0) = 0
   LET  mag01(1) = MATRIX_A                  !           static   unsigned   long
mag01[2]={0x0UL, MATRIX_A};
   REM /* mag01[x] = x * MATRIX_A  for x=0,1 */


   IF mti >= P THEN                          !     if (mti >= P) { /* generate P
words at one time */
      DECLARE NUMERIC kk                     !        int kk;


      IF mti = P + 1 THEN                    !           if (mti == P+1)    /* if
init_genrand() has not been called, */
         LET  y = init_genrand(5489)         !                init_genrand(5489UL);
/* a default initial s is used   */
      END IF
      FOR kk=0 TO P-Q-1                      !        for (kk=0;kk<P-Q;kk++) {
         LET     y   =   Xor32(And32(mt(kk)   ,   UPPER_MASK)   ,   And32(mt(kk+1)   ,
LOWER_MASK))
         !                                                !                y =
(mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
         LET   mt(kk)  =  Xor32(Xor32(mt(kk+Q)  ,  SR32U(y  ,  1))  ,  mag01(And32(y  ,
1)))
         !                                   !           mt[kk] = mt[kk+Q] ^ (y
>> 1) ^ mag01[y & 0x1UL];
      NEXT kk                                !        }
      FOR kk=kk TO P-2                         !       for (;kk<P-1;kk++) {

         LET    y   =   Xor32(And32(mt(kk)    ,   UPPER_MASK)    ,   And32(mt(kk+1)    ,
LOWER_MASK))

         !                                     !                                   y   =
(mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);

         LET   mt(kk)  =  Xor32(Xor32(mt(kk+Q-P) , SR32U(y , 1)) , mag01(And32(y ,
1)))

         !                                     !             mt[kk]  = mt[kk+(Q-P)]  ^
(y >> 1) ^ mag01[y & 0x1UL];

      NEXT kk                                  !        }

      LET  y = Xor32(And32(mt(P-1) , UPPER_MASK) , And32(mt(0) , LOWER_MASK))

      !                                                            !        y  =  (mt[P-
1]&UPPER_MASK)|(mt[0]&LOWER_MASK);

      LET  mt(P-1) = Xor32(Xor32(mt(Q-1) , SR32U(y , 1)) , mag01(And32(y , 1)))

      !                                        !     mt[P-1]  = mt[Q-1]  ^  (y >> 1) ^
mag01[y & 0x1UL];


      LET  mti = 0                             !       mti = 0;

   END IF                                      !    }


   LET  y = mt(mti)

   LET  mti = mti + 1                          !    y = mt[mti++];


   REM /* Tempering */

   LET  y = Xor32(y , SR32U(y , 11))           !    y ^= (y >> 11);

   LET  y = Xor32(y , And32(SL32U(y , 7) , BVAL("9d2c5680" , 16)))

   !                                           !    y ^= (y << 7) & 0x9d2c5680UL;

   LET  y = Xor32(y , And32(SL32U(y ,15) ,BVAL("efc60000" , 16)))

   !                                           !    y ^= (y << 15) & 0xefc60000UL;

   LET  y = Xor32(y , SR32U(y , 18))           !    y ^= (y >> 18);


   LET  genrand = y                            !    return y;

END FUNCTION                                   !}
```

```
REM
/*********************************************************************************/

REM /* generates a random number on [0,0x7fffffff]-interval */

FUNCTION genrand_31                          !long genrand_31(void)  {

   LET  genrand_31 = SR32U(genrand , 1)     !    return (long)(genrand()>>1);

END FUNCTION                                !}
```

B.5 Linear congruential method

B.5.1 General

B.5.1.1 Usage

Linear congruential methods are widely used in software since they combine economy of use of memory with rapid execution. However, they have a relatively short period and are consequently not sufficiently random, particularly for generating random multi-dimensional sequences.

B.5.1.2 Definition

Most linear congruential methods generate pseudo-random number sequences X_1, X_2, ... by using the following recurrence relationship.

$$X_n = \mathrm{mod}(aX_{n-1} + c; m) \quad n = 1, 2, ...$$

where a and m are positive integers and c is a non-negative integer.

Once the values of the parameters a, m and c have been decided, the linear congruential method is determined; moreover, if the seed X_0 is given, the generated number sequence is determined.

NOTE 1 The meaning of the recurrence relationship is as follows. Calculate $aX_0 + c$ by using seed X_0 and divide the result by m. The residue is X_1. Next, calculate $aX_1 + c$ and divide the result by m, and the residue is X_2. This procedure is repeated as many times as required.

NOTE 2 The value of n for which $X_n = X_0$ for the first time is called the period of the sequence.

B.5.1.3 Method of deciding parameter values

A good pseudo-random number sequence cannot be obtained if the values of a, m and c are determined arbitrarily. Therefore, these should be decided on the following basis.

Because m is the upper limit of a period of the number sequence obtained by the linear congruential method, m should be set as large as possible. Hence, using for example 32-bit computers, it is recommended to set $m = 2^{32}$ or $2^{31} - 1$.

For increment c, there is no strict criterion. However, the periods of generated number sequences may be different, according to whether a criterion is set to zero or a positive integer.

As for the multiplier a, a value that provides good results in combination with the chosen values of m and c should be used (see Table B.1).

NOTE In the case where m is an integer power of 2 and c is specified to be 0, the period is not more than $m/4$. If c is an odd number, the period becomes m.

B.5.1.4 Example of parameters

For 32-bit computers, one of the sets of parameters from Table B.1 should be used.

Table B.1 — Examples of parameters used in the linear congruential method

Row number	A	c	M
1	1 664 525	*	2^{32}
2	1 566 083 941	0	2^{32}
3	48 828 125	0	2^{32}
4	2 100 005 341	0	$2^{31} – 1$
5	397 204 094	0	$2^{31} – 1$
6	314 159 369	0	$2^{31} – 1$

NOTE 1 The symbol * indicates that any odd number may be used.

NOTE 2 Using the parameters of row 1, all integers in the range 0 to ($2^{32} – 1$) are generated.

NOTE 3 Using the parameters of row 2 or row 3, the set of generated numbers is $4i + 1$ for $i = 0$, 1, ..., ($2^{30} – 1$), or $4i + 3$ for $i = 0$, 1, ..., ($2^{30} – 1$), which depends on the seed X_0.

NOTE 4 Using the parameters of rows 4, 5 or 6, all the positive integers between 1 to ($2^{31} – 2$) are generated.

NOTE 5 When not many bits are required, they should be extracted from the upper bits of the random number, and lower bits should not be used.

B.5.2 Program code for the linear congruential method

B.5.2.1 General

The C language implementation of the linear congruential method, which is in accordance with ISO/IEC 9899, is given below. It is composed of two types of programs, one for the case where $m = 2^{32}$, the other for the case where $m = 2^{31} – 1$. These cases are consistent with the recommendations of B.5.1.

B.5.2.2 Case of $m = 2^{32}$

Every time the function lcong32( ) is called, it returns an integer random number between zero and ($2^{32} – 1$) inclusive. The result is returned of type “unsigned long”. Every time the function lcong32_31( ) is called, it generates an integer random number between zero and ($2^{31} – 1$) inclusive. The result is returned of type “long”. The initialization function init_lcong32 (unsigned long seed) executes initialization so that a non-negative integer of type “unsigned long” is set as the seed. If the addend c is 0 and the original seed X_0^* is odd, then X_0 can be set as $X_0 = X_0^*$. However, when an even original seed X_0* is given in the case $c = 0$, one is added to the original seed to obtain the seed X_0, i.e. $X_0 = X_0^* + 1$. In other cases, mod(X_0*; m) is used as X_0.

The multiplier and the addend are changed by changing the definitions of MULTIPLIER and INCREMENT in each program. The random number sequence is restarted by using the output of lcong32( ) as the argument of initialization function init_lcong32(seed).

B.5.2.3 Case of $m = 2^{31} – 1$

Every time the function lcong31( ) is called, it returns an integer random number between 1 and ($2^{31} – 2$) inclusive. The result is returned of type “long”. The initialization function init_lcong31 (unsigned long seed) executes initialization so that a non-negative integer is set as the seed. As with the parameters in Table B.1, the addend of c are always 0. Therefore, the seed X_0 should not be 0. However, if $X_0^* = 0$, a special number (19 660 809) is used instead as the seed X_0. In other cases, mod(X_0*; m) is used as X_0.

In order to change the multiplier, the definition of MULTIPLIER should be changed in the program.

```
/*****************************************************************
 C code : Linear Congruential
*****************************************************************/

/*********************************************************
 Part 1. Modulus = 2^32
*********************************************************/

#define MULTIPLIER 1664525UL
#define INCREMENT 1UL

static unsigned long state32 ;

unsigned long lcong32( void )
{
  state32 = ( state32 * MULTIPLIER + INCREMENT ) & 0xFFFFFFFFUL;
  return state32;
}

long lcong32_31( void )
{
  state32 = ( state32 * MULTIPLIER + INCREMENT ) & 0xFFFFFFFFUL;
  return state32>>1;
}

void init_lcong32(unsigned long s)
{
/* seed should be odd when increment == 0 */
     if ( (INCREMENT==0) && (s%2 == 0) ){
         s++ ;
     }
     state32 = s ;
}

/*********************************************************
 Part 2. Modulus = 2^31-1 = 2147483647
*********************************************************/

#undef MULTIPLIER
```

```
#undef INCREMENT
#undef NBIT

#define NBIT 15
#define MASK  ( (1<<NBIT)-1)
#define MASK2 ( (1<<2*NBIT) -1)
#define MULTIPLIER 2100005341UL
#define MULTIPLIER_LO (MULTIPLIER & MASK)
#define MULTIPLIER_HI (MULTIPLIER >> NBIT)
static unsigned long state31 ;

long lcong31 ( void )
{
 unsigned long xlo, xhi ;
 unsigned long z0, z1, z2 ;

 xlo = state31 & MASK ;
 xhi = state31 >> NBIT ;
 z0  = xlo * MULTIPLIER_LO ;   /* 15bit * 15bit => 30bit */
 z1  = xlo * MULTIPLIER_HI
    + xhi * MULTIPLIER_LO ;   /* 15bit * 16bit * 2 => 32bit */
 z2  = xhi * MULTIPLIER_HI ;   /* 16bit * 16bit => 32bit */
 z0 += (z1 & MASK) << NBIT ;
 z2 += (z1 >> NBIT) + (z0 >> (2*NBIT)) ;
 z0  = (z0 & MASK2) | ((z2&1) << (2*NBIT)) ;
 z2 >>=1 ;
 state31 = z0 + z2 ;
 /* This should not exceed 2*0x7fffffffUL */
 if (state31>=0x7fffffffUL) state31 -= 0x7fffffffUL ;
 return (long) state31 ;
}
void init_lcong31 (unsigned long s)
{
 /* seed should not be 0 */
 if ( s == 0UL ) s=19660809UL ;
 s = s % 0x7fffffffUL ;
 state31 = s ;
}
```

NOTE 1 The Full Basic code of the linear congruential method is shown for information comparing with the corresponding C code as follows.

```
REM /*********************************************************

REM   BASIC code : Linear Congruential

REM *********************************************************/

REM

REM /*********************************************************

REM    Part 1. Modulus = 2^32

REM *********************************************************/


OPTION BASE 0


REM
/********************************************************************************/

DECLARE NUMERIC MULTIPLIER

LET  MULTIPLIER = 1664525                    !#define MULTIPLIER 1664525UL

DECLARE NUMERIC INCREMENT

LET  INCREMENT = 1                           !#define INCREMENT  1UL


DECLARE NUMERIC state32                      !static unsigned long state32;


REM
/********************************************************************************/

FUNCTION lcong32                             !unsigned long lcong32u( void )     {

   LET  state32 = And32((state32 * MULTIPLIER) + INCREMENT , MskF_f)

   !                                         !  state32 = ( state32 * MULTIPLIER +
INCREMENT ) & 0xFFFFFFFFUL;

   LET  lcong32 = state32                    !  return state32;

END FUNCTION                                 !}


REM
/********************************************************************************/

FUNCTION lcong32_31                         !long lcong32( void )   {
```

```
!                                         !  state32 = ( state32 * MULTIPLIER +
INCREMENT ) & 0xFFFFFFFFUL;

   LET  lcong32_31 = SR32U(lcong32 , 1)   !  return state32>>1;

END FUNCTION                              !}


REM
/*******************************************************************************/

FUNCTION init_lcong32(s)                  !void init_lcong32(unsigned long s) {

   REM /* seed should be odd when increment == 0 */

   IF (INCREMENT = 0) AND (REMAINDER(s , 2) = 0) THEN

   !                                      !  if ( (INCREMENT==0) && (s%2 == 0) )
{

      LET  s = s + 1                      !    s++;

   END IF                                 !  }

   LET  state32 = s                       !  state32 = s;

END FUNCTION                              !}


REM /**********************************************************

REM   BASIC code : Linear Congruential

REM **********************************************************/

REM

REM /**********************************************************

REM    Part 2. Modulus = 2^31-1 = 2147483647

REM **********************************************************/


OPTION BASE 0


DECLARE NUMERIC NBIT

LET  NBIT = 15                            !#define NBIT 15

DECLARE NUMERIC MASK

LET  MASK = SL32U(1 , NBIT) - 1                !#define MASK  ((1<<NBIT)-1)
```