Journal of Computer Science

Original Research Paper

Twister Generator of Random Normal Numbers by Box-Muller Model

[1]Aleksei F. Deon and [2]Yulian A. Menyaev

[1]Department of Information Systems and Computer Science, N.E. Bauman Moscow State Technical University, Moscow, Russia
[2]Winthrop P. Rockefeller Cancer Institute, University of Arkansas for Medical Sciences, Little Rock, AR, USA

Article history
Received: 07-10-2019
Revised: 21-12-2019
Accepted: 09-01-2020

Corresponding Author:
Yulian A. Menyaev
Winthrop P. Rockefeller Cancer Institute, University of Arkansas for Medical Sciences, Little Rock, AR, USA
Email: yamenyaev@uams.edu

Abstract: Twisting generators of the pseudorandom normal variables can use uniform random sequences as a basis. However, such technique could provide poor quality result in cases where the original sequences have insufficient uniformity or skipping of random values. This work offers a new approach for creating the random normal variables using the Box-Muller model as a basis together with the twisting generator of uniform planes. The simulation results confirm that the random variables obtained have a better approximation to normal Gaussian distribution. Moreover, combining this new approach with the tuning algorithm of basic twisting generation allows for a significantly increased the length of created sequences without using any additional random access memory of the computer.

Keywords: Pseudorandom Number Generator, Stochastic Sequences, Congruential Numbers, Twister Generator, Normal Plane Generator

Introduction

The direction of Gaussian Random Number Generator (GRNG) realizes the process of creating the random variables $\zeta \in Z$ with the function of normal distribution $F_Z(\zeta)$. Since the random variable $z \in Z(m,\sigma)$ with arbitrary first moments $m = E_1(\zeta \in Z)$, $D = E_2(\zeta \in Z) = E_1(\zeta^2)$ could be reduced to the standard random normal variable $\xi \in \Xi(m = 0, \sigma = 1)$, then standard GRNGs are usually used to ensure a normal Gaussian probability distribution:

$$F_{\Xi}(\xi \in \Xi) = \int_{-\infty}^{\xi} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx \quad (1)$$

Normal GRNGs are widely used in mathematical studies (Halim *et al*., 2012; Neugebauer *et al*., 2019), systems of message communication (Lee *et al*., 2006; Martino *et al*., 2012; Zhou *et al*., 2014), designing of computer games (Sukajaya *et al*., 2012), complicated technical systems (Malik *et al*., 2011; Paraskevakos and Paliouras, 2011), models for financial analysis (Sauer, 2012), biological studies (Menyaev *et al*., 2013; 2016), development of medical devices (Menyaev and Zharov, 2005; 2006a; 2006b) and as well as in many other applied fields.

There are many different ways to implement GRNG. Among all of them the generators using the Box-Muller model (Box and Muller, 1958) are applied widely. In this type of generation the random variables R are created using uniformly distributed random values u and v by one of the following two expressions:

$$R(u,v) = \sqrt{-2\ln u}\cos(2\pi v) \quad (2)$$

$$R(u,v) = \sqrt{-2\ln u}\sin(2\pi v) \quad (3)$$

In the most accessible and widespread form of use for this generator is given in Wikipedia (Wikipedia.org/wiki/Box-Muller_transform), in which the program code of generator is presented in the programming language *C*. This code contains the main generation cycle in the following form:

```
if (!generate)
   return z1 * sigma+mu;
double u1, u2;
do
 {
   u1 = rand() * (1.0/RAND_MAX);
   u2 = rand() * (1.0/RAND_MAX);
 }
while ( u1 <= epsilon );
z0 = sqrt(-2.0 * log(u1)) * cos(two_pi * u2);
z1 = sqrt(-2.0 * log(u1)) * sin(two_pi * u2);
```

DOI: 10.3844/jcssp.2020.1.13

Here the random variables *z*0 and *z*1 correspond to expressions (2) and (3). As a generator of uniform random variables the standard function *rand()* is used, which might be applied, for example, in any version of *Microsoft Visual Studio*. Directly checking the uniformity of function *rand()* shows that the random variables obtained do not have sufficient quality in terms of uniqueness properties, i.e. both skipping and repeating of the generated variables is present. Therefore, the randomness of *z*0 and *z*1 is far from perfect. Let's consider the uniformity quality of function *rand()* more in detail in accordance with simple a two-stage algorithm.

At the first stage, it is necessary to determine experimentally the minimum and maximum (*min* and *max* respectively) values of function *rand()* in the range of the generated whole random variables. At the second stage, let's perform the simplest one-time generation in this range of [*min*, *max*] and that is together with counting how many times each value is created by using counters. If generator *rand()* is uniform, then it has to demonstrate in counters a single generation for each random variable, since the size of [*min*, *max*] range exactly corresponds to the amount of the single-valued generation of random variables. If the values of the counters are different, it means that function *rand()* isn't uniform. In this case, the generator of normal values, which follows expressions (2) or (3), provides insufficient quality of distribution of the random normal variables.

Below is the program code for the first aforementioned stage which is written in *Microsoft Visual Studio*. The programming language uses *C#* dialect. Similar results can be obtained in the language of historical *C* or object-oriented *C++* one. The function *rand()* in historical *C* produces the random values in the range [0: 0*x*7*FFF*]. Compare to this, the range of function *Random.Next()* in *C#* is extended significantly. This could be obtained experimentally using the following code below.

```
namespace P050101
{ class cP050101
  { static void Main(string[] args)
    { for (int w = 16; w <= 31; w++)          // bit length
      { uint N1 = 0xFFFFFFFF >> (32 - w);     // max
        Console.Write("w = {0} N1 = {1,12} ", w,
                      N1);
        uint min = 0xFFFFFFFF;
        uint max = 0;
        Random r = new Random();
        for (uint i = 0; i <= N1; i++)
        { uint v = (uint)(r.Next());
          if (v < min) min = v;
          else if (max < v) max = v;
        }
        Console.WriteLine(
          "min = {0,9:X} max = {1,9:X}", min, max);
      }
      Console.WriteLine("The test is over");
      Console.ReadKey();                  //result viewing
    }
  }
}
```

After starting program *P050101*, the following result appears on the monitor. Parameter *w* specifies the bit length of the random variables. The value of *N*1 determines the maximum whole decimal number in the corresponding range. The *min* and *max* values are in hexadecimal form. They show the real values achieved in each phase of the experiment:

```
w = 16  N1 =   65535  min = 11EC4   max = 7FFEEDD5
w = 17  N1 = 131071   min = 11EA5  max = 7FFFDBA1
w = 18  N1 = 262143  min =   142C   max = 7FFFF0A0
w = 19  N1 = 524287  min =   1293   max = 7FFFFC07
w = 20  N1 = 1048575  min =   877   max = 7FFFFC07
w = 21  N1 = 2097151  min =   2A7   max = 7FFFF7C1
w = 22  N1 = 4194303  min =   309   max = 7FFFFF4D
w = 23  N1 = 8388607  min =   54   max = 7FFFFFCD
w = 24  N1 = 16777215 min =   A4   max = 7FFFFF7A
w = 25  N1 = 33554431 min =   3   max = 7FFFFFB7
w = 26  N1 = 67108863 min =   1A   max = 7FFFFFEC
w = 27  N1 = 134217727     min = 1   max = 7FFFFFF9
w = 28  N1 = 268435455     min = 3   max = 7FFFFFEE
w = 29  N1 = 536870911     min = 1   max = 7FFFFFFC
w = 30  N1 = 1073741823   min = 1   max = 7FFFFFFE
w = 31  N1 = 2147483647   min = 0   max = 7FFFFFFE
```

This listing experimentally confirms that integers with a bit length up to 31 bits long, for example, are generated from the range [0: 0*x*7*FFFFFFF*]. So, function *Random.Next()* provides a sequence of the random variables having a length of 31 bits.

The second stage of the uniformity check has to contain the counters that take into account the issue of how many times each number from range [0: 0*x*7*FFFFFFF*] is generated. However, this can't be done directly on a computer with a 32-bit data bus, since in this case there is no space in the computer's Random Access Memory (RAM) for the operating system and this program itself. So, in order to perform the second stage, the presented below program *P050102* uses an array of 2^{27} counters for each of the intervals of random variables [0: $1\cdot 2^{27}$-1], then [$1\cdot 2^{27}$: $2\cdot 2^{27}$-1] and so on until $2^4 = 16$ times to the interval [$15\cdot 2^{27}$: $2^4 \cdot 2^{27}$-1] is reached. Thus, based on this it would be possible to consider in detail the values for counters in the range [0: 2^{31}-1] = [0: 0*x*7*FFFFFFF* - 1] of random variables. At each of the 16

DOI: 10.3844/jcssp.2020.1.13

iterations, the generation of 2^{31} random numbers is carried out together with adding the amount of the corresponding generations in counters of the next subband with length 2^{27} of random variables:

```
namespace P050102
{ class cP050102
  { static void Main(string[] args)
    { int w = 31;                    //generation for bit length
      uint N1 = 0xFFFFFFFF >> (32 - w);          //max
      Console.WriteLine("w = {0} N1 = {1}", w, N1);
      int cw = 27;                          //interval bit length
      int nc = 1 << cw;   //amount of numbers & counters
      Console.WriteLine(
            "cw = {0} nc = {1} nc = 0x{1:X}", cw, nc);
      int[] c = new int[nc];                //array of counters
      Random r = new Random();       //generation object
      int m = 1 << (w - cw);          // quantity of intervals
      for (int i = 0; i < m; i++)
      { Console.Write("i = {0,2} ", i);             //interval
        int n1 = i * nc;                //bottom counter edge
        int n2 = (i + 1) * nc - 1;                  //top edge
        Console.WriteLine(
              "n1 = {0,10:X} n2 = {1,10:X} ",
            n1, n2);
        for (int j = 0; j < nc; j++) c[j] = 0;
        for (uint k = 0; k <= N1; k++)
        { uint v = (uint)r.Next();
          if (n1 <= v && v < n2) c[v - n1]++;
        }
        int q0 = 0, q1 = 0, q2 = 0, q3 = 0;
        for (int j = 0; j < nc; j++)
        { if (c[j] == 0) q0++;
          else if (c[j] == 1) q1++;
          else if (c[j] == 2) q2++;
          else q3++;
        }
        Console.Write("       ");
        Console.Write(
             "q0 = {0,10} q1 = {1,10} ", q0, q1);
        Console.WriteLine(
             "q2 = {0,10} q3 = {1,10}", q2, q3);
      }
      Console.ReadKey();                    //result viewing
    }
  }
}
```

After running the program *P050102*, the following listing appears on the monitor:

```
w = 31 N1 = 2147483647
cw = 27 nc = 134217728 nc 0x8000000
i = 0    n1 = 0          n2 = 7FFFFFF    q0 = 49382360    q2 = 24682358    q3 = 10778132
i = 1    n1 = 8000000    n2 = 7777777    q0 = 49380286    q2 = 24693721    q3 = 10770790
i = 2    n1 = 10000000   n2 = 17FFFFFF   q0 = 49372134    q2 = 24684975    q3 = 10783667
i = 3    n1 = 18000000   n2 = 1FFFFFFF   q0 = 49379005    q2 = 24692946    q3 = 10779396
i = 4    n1 = 20000000   n2 = 27FFFFFF   q0 = 49375891    q2 = 42688306    q3 = 10777756
i = 5    n1 = 28000000   n2 = 2FFFFFFF   q0 = 49391391    q2 = 24685803    q3 = 10774012
i = 6    n1 = 30000000   n2 = 37FFFFFF   q0 = 49376688    q2 = 24693261    q3 = 10777766
i = 7    n1 = 38000000   n2 = 3FFFFFFF   q0 = 49381811    q2 = 24687840    q3 = 10773706
i = 8    n1 = 40000000   n2 = 47FFFFFF   q0 = 49368176    q2 = 24689824    q3 = 10777062
i = 9    n1 = 48000000   n2 = 4FFFFFFF   q0 = 49369637    q2 = 24691240    q3 = 10775962
i = 10   n1 = 50000000   n2 = 57FFFFFF   q0 = 49380081    q2 = 24682605    q3 = 10770596
i = 11   n1 = 58000000   n2 = 5FFFFFFF   q0 = 49378595    q2 = 24680123    q3 = 10784203
i = 12   n1 = 60000000   n2 = 67FFFFFF   q0 = 49376383    q2 = 24689112    q3 = 10779434
i = 13   n1 = 68000000   n2 = 6FFFFFFF   q0 = 49368619    q2 = 24689890    q3 = 10772054
i = 14   n1 = 70000000   n2 = 77FFFFFF   q0 = 49381570    q2 = 24683658    q3 = 10775170
i = 15   n1 = 78000000   n2 = 7FFFFFFF   q0 = 49376263    q2 = 24690133    q3 = 10779016
```

An analysis of counters in this listing shows that in each interval there is the following: some random variables are missed (counter $q0$), others are generated once ($q1$), the remaining variables are repeated twice or more times ($q2$+$q3$). These results allow for a conclusion that generator *Random.Next()* from *C#* language and its predecessor *rand()* from historical *C* one for which the printouts are the same, unfortunately they are unable to provide a sufficient quality of uniformity for generation of random normal variables in accordance with (2) or (3) using the Box-Muller technique.

In connection with all this above, the purpose of this article is to create a novel high-quality generator based on the Box-Muller transformation together with a technique of an absolutely uniform random number generation which we proposed and explored recently.

Theory

In modern probability theory, the Kolmogorov's axiomatics (Kolmogorov, 1968) uses the fact of one-to-one correspondence between the random variable and the

probability distribution function. In the continual domain (Kolmogorov and Fomin, 1999) of finding the random variable $\xi \in \Xi$, the distribution function $F_\Xi(\xi)$ is bounded by the range [0,1]. From the fact of uniqueness it follows that if any value h of the distribution function $F_\Xi(\xi_h)$ is given, then the value of the random variable ξ_h can be obtained as the inverse transformation of function $\xi_h = F_\Xi^{-1}(h)$. By the definition of distribution function, the continuity property guarantees a strict ordering of values of the random variables. Consequently, by specifying uniform complete random (even continual) values of the distribution function, it is possible to obtain identically the random values of this distribution. So, this main mathematical model contains the bases for constructing the generators of random variables in correspondence with given functions of their distribution. Let's take this statement into account for developing a generator of the random normal variables.

By the definition, function $F_\Xi(\xi)$ of normal distribution of the random variables $\xi \in \Xi$ has the form as it is presented in expression (1) above. This expression allows calculating directly the random variable ξ using well-known methods of integrating, among of which the Darboux-Riemann technique is standing out as most accurate. However, this approach may result in the increase of calculation time, which is proportional to the given accuracy of integrating. In some tasks of the substantial trials, such a time limit could be a significant restriction.

The distinctive solution was proposed in the famous work by Box and Muller (1958), which uses an approach analogous to the Rayleigh distribution algorithm (Feller, 2008; Gnedenko, 1998). In this technique it is given the joint random variable $\vartheta =< s, t >\in \Theta$ of independent variables $s \in S$ and $t \in T$ with the same normal distribution functions. Assuming that both random variables s and t are coincide with the random variable $\xi \in \Xi$, then their normal distribution density in accordance with expression (1) has the following form:

$$f_S(s) = f_T(t) = f_\Xi(\xi) = \frac{1}{\sqrt{2\pi}} e^{-\frac{\xi^2}{2}} \tag{4}$$

The probability distribution of the joint random variable $\vartheta =< s, t >\in \Theta$ on the Descartes plane $S_\vartheta \times T_\vartheta$ is defined by the following expression:

$$F_\Theta\left(\vartheta =< S_\vartheta, T_\vartheta >\right) = \int_{-\infty}^{S_\vartheta}\int_{-\infty}^{T_\vartheta} f_\Theta(s,t)\cdot dsdt \tag{5}$$

Taking into account the independence of the random variables s and t in (4), the joint probability density $f_\Theta(s, t)$ is determined by the following multiplication:

$$f_\Theta(s,t) = f_S(s)\cdot f_T(t) = \\ = \frac{1}{\sqrt{2\pi}} e^{-\frac{s^2}{2}} \cdot \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} = \frac{1}{2\pi} e^{-\frac{s^2+t^2}{2}} \tag{6}$$

Substituting (6) into (5), the following expression appears:

$$f_\Theta\left(\vartheta =< S_\vartheta, T_\vartheta >\right) = \int_{-\infty}^{S_\vartheta}\int_{-\infty}^{T_\vartheta} \frac{1}{2\pi} e^{-\frac{s^2+t^2}{2}} \cdot dsdt \tag{7}$$

Direct calculation of the probability function $F_\Xi(S,T)$ by expression (7) could be performed by any methods of the numerical integrating. However, this might take a long time. At the same time, it is possible to reduce the amount of calculations because expression (7) may be redirected to polar coordinates. Formally, the sum $r_\vartheta^2 = s^2 + t^2$ corresponds to the length of the radius-vector $\vartheta =< s, t >$:

$$r_\vartheta = \sqrt{s^2 + t^2} \tag{8}$$

In this case, to each radius-vector $\vartheta =< s, t >$ in space $S\times T$ corresponds a vector $r \in R$ at an angle $\varphi \in \Phi = [0: 2\pi]$ in the polar coordinates of space $R\times\Phi$. Their lengths r and r_ϑ are equal and the coordinates of vectors ϑ and r are interrelated:

$$\begin{aligned} r &= r_\vartheta \\ s &= r\cdot\sin\varphi \\ t &= r\cdot\cos\varphi \end{aligned} \tag{9}$$

This geometric representation allows interpreting expression (7) as a distribution function for the length of the radius-vector ϑ in the Descartes space $S\times T$. The same corresponds to the distribution function $F_r(R,\Phi)$ of vector r in the polar space $R\times\Phi$. The transition $F_r(R,\Phi) = F_\Theta(S,T)$ could be performed with the help of Jacobian J, which allows replacing the multiplication of differentials $ds\cdot dt$ by analogues $dr\cdot d\varphi$:

$$F_r(R,2\pi) = \int_0^R\int_0^{2\pi} \frac{1}{2\pi} e^{-\frac{r^2}{2}} \cdot J\cdot drd\varphi \tag{10}$$

The form of Jacobian J is defined as follows:

$$J = \begin{vmatrix} \frac{\partial s}{\partial r} & \frac{\partial s}{\partial \varphi} \\ \frac{\partial t}{\partial r} & \frac{\partial t}{\partial \varphi} \end{vmatrix} = \begin{vmatrix} \cos\varphi & -r\cdot\sin\varphi \\ \sin\varphi & r\cdot\cos\varphi \end{vmatrix} = r\cdot\cos^2\varphi + r\cdot\sin^2\varphi = r \tag{11}$$

DOI: 10.3844/jcssp.2020.1.13

Substituting (11) into (10) allows getting this:

$$F_r\left(R,2\pi\right)=\int_0^R\int_0^{2\pi}\frac{1}{2\pi}e^{-\frac{r^2}{2}}r\cdot drd\varphi=$$
$$=\int_0^R re^{-\frac{r^2}{2}}dr\cdot\frac{1}{2\pi}\int_0^{2\pi}d\varphi=\int_0^R re^{-\frac{r^2}{2}}dr=1-e^{-\frac{R^2}{2}} \quad (12)$$

An important result of expression (12) is the fact that using two random variables *S* and *T*, it is possible to calculate explicitly in polar coordinates the normal distribution $1-e^{-R^2/2}$ and that is without applying the methods of the numerical integrating like it is in expression (1) for the rectangular Descartes coordinates. In this case, the Kolmogorov's axiomatics main property (Gnedenko, 1998) of one-to-one correspondence of a random variable to its distribution function can be applied. Exactly this property is used in the generator of the random normal variables, which uses the Box-Muller technique. The point here is: it might take a complete generator of uniform random variables for the specific generations of values of a function of the random normal variables and then, for each value, the random normal variables will be calculated using the inverse function. The uniqueness of Kolmogorov's axiomatics guarantees the correct result. For this technology the result (12) is best suited, since the vector length in form of the random variable is the same in both spaces, i.e. *S*×*T* and *R*×Φ.

In (1) the centered random variable ξ in space *S*×*T* can take both positive and negative values. However, in the polar space *R*×Φ the random variable *r* has only a positive value. To somehow smooth this insoluble contradiction, the Box-Muller model uses an artificial technique, which introduces the second uniform generator for angle φ on the circular interval [0: 2π] having radius *r* = *R*. For this, the elementary trigonometric transformations are suitable, for example, like form cos φ in (2) or form sin φ in (3). The error is negligible in this case, but positive and negative random normal variables are obtained perfectly. This approach is close to the different decisions used in probability theory in case of situation when an error of the modeling isn't important significantly for applied tasks.

Let's demonstrate the model of such a technique. First of all, it is necessary to point out now that by property of any distribution function, its cumulative value for all the random variables is singular:

$$F_r\left(R=\infty,2\pi\right)=1-e^{-\frac{\infty^2}{2}}=1 \quad (13)$$

Since sequence of the generated random variables is random but contains all the random variables *u*' for $F_r(R,2\pi)$, the complete sequence of corresponding realizations of the distribution function $F_r(u',2\pi)$ is the random sequence as well. It contains both, values $F_r(u',2\pi)$ and values $F_r(u = 1\text{-}u', 2\pi)$ because of accordance to property (13). It is obvious that order of an enumeration of the random variables *u* and *u*' in functions (12) is not of significant importance because of their randomness and cumulativity.

Now with the help of the complete uniform generator it is necessary to obtain uniform random variable *u* for expression (12) on interval [0: 1]. Since the axiomatics of probability theory provides the appearance of all the random variables in complete sequences of the observed events, then the following variant for expression (12) could be used:

$$u=1-u'=1-e^{-\frac{R^2}{2}}. \quad (14)$$

Expression (14) allows calculating the inverse function $R=F_r^{-1}\left(u,2\pi\right)$:

$$R=F_r^{-1}\left(u,2\pi\right)=\sqrt{-2\ln u} \quad (15)$$

At this step, it is required to introduce a technological correction $u\in U$ = (0: 1]. It means that random variable *u* = 0 has to be excluded from (15) since ln0 is not a subject to calculation.

Further, it is necessary to determine the sign of the random variable *R*, since both values of $R=\pm\sqrt{-2\ln u}$ are permissible. For this purpose, in the Box-Muller model the sign-factor of trigonometric functions (2) and (3) is used. For the angular random variable φ to have uniform distribution on the circular interval [0, 2π], it is proposed to use the second complete uniform generator of the random variable *v*. However, on the circular interval the points 0 and 2π coincide, therefore one of them should be abandoned. Since in generator (15) the random variable *u* = 0 isn't used, it is preferable to choose the half-open circular interval (0, 2π]. Assuming that the second generator creates the random variables $v \in$ (0: 1], the value of the random angle is obtained as the following:

$$\varphi=2\pi\cdot v \quad (16)$$

Collecting together the generators in expressions (15) and (16), the final views of generator for the random normal variable *z* look as follows:

$$z=R\cdot\cos\varphi=\sqrt{-2\ln u}\cos\left(2\pi\cdot v\right) \quad (17)$$

Or the same in similar form:

$$z=R\cdot\sin\varphi=\sqrt{-2\ln u}\sin\left(2\pi\cdot v\right)$$

At this point, let's finish the theoretical considerations. Forms of expression (17) with taking into account the random Descartes plane $S \times T$ allows creating the required generator of the random normal variables according to the Box-Muller model.

Construction and Results

In expression (10), plane $R \times \Phi = [0,R] \times [0,2\pi]$ is used to calculate the distribution function $F_r(R,2\pi)$. According to (13), the maximum value is $F_{r\max}(\infty,2\pi) = 1$. Consequently, in the model (17) there is $u \in BMU = (0: 1]$. At the same time, according to (16) another independent complete uniform generator creates the random variables $v \in BMV = (0: 1]$. To realize the coordinated work of these generators, it is necessary to use some complete generator of the real random plane.

In our previous work (Deon and Menyaev, 2016a), we simulated absolutely all possible uniform sequences, but the problems of designing the generator itself weren't considered there. In continuation of that work, then in (Deon and Menyaev, 2016b) we proposed a twister generator of the complete uniform random variables using the technology of a twisting array. In order to exclude the influence of the twisting array on the computer RAM, we have perfected the previous development by proposing a generator of uniform twisting sequences of arbitrary size but without twisting array (Deon and Menyaev, 2017; 2019). The technology of this generator is the basis of generator *cDeonYuliPlaneTwist32D* (Deon and Menyaev, 2018), which creates the random planes $N \times N = 2^w \times 2^w = \left[\overline{0:2^w-1}\right] \times \left[\overline{0:2^w-1}\right]$ using whole random variables of length w bits. Now the resulting random whole points on the Descartes plane $N \times N$ should be transformed into random real points on plane $U \times V = (0: 1] \times (0: 1]$. For every such point $<u, v>$ there is a correspondence of the random normal variable z according to (17). Now let's figure out how it all works.

In the base class *cDeonYuliPlaneTwist32D* (Deon and Menyaev, 2018) two variables u and v having the length of w bits are created from range $[0: 2^w - 1]$. Then these quantities are transformed into random real variables du and dv as follows. At each half-open intervals BMU and BMV there are $N = 2^s$ segments of length:

$$d_{du} = d_{dv} = d = \frac{1}{N} \tag{18}$$

The first initial segment corresponds to the subinterval $du_1 = dv_1 = (0: 1 \cdot d]$, the next one to $du_2 = dv_2 = (1 \cdot d: 2 \cdot d]$ and the last to $du_N = dv_N = [(N - 1) \cdot d: N \cdot d]$ accordingly. Note that index k on the right side of each subinterval $du_k = dv_k$ is the marker of all subintervals $k \in \left[\overline{1,N}\right] = \left[\overline{1,2^w}\right]$. The main thing here is that $k \neq 0$ and this is perfectly suitable for the further calculations in (17).

Below is class *cDeonYuliBMNormalTwist32D*, in which the random normal variables are created on the random half-open plane $BMU \times BMV = (0: 1] \times (0: 1]$, for which a square grid with step d (18) contains the twisting random variables. Class *cDeonYuliBMNormalTwist32D* is derived over the base class of twisting planes *cDeonYuliPlaneTwist32D* (Deon and Menyaev, 2018). An example of generation of the random normal variables is given later in program *P050301*:

```
using nsDeonYuliPlaneTwist32D;
namespace nsDeonYuliBMNormalTwist32D
{ class cDeonYuliBMNormalTwist32D :
cDeonYuliPlaneTwist32D
  { public double d;                          //interval length
    public uint u;          //integer random number along U
    public uint v;          //integer random number along V
    public double bmu;//real random number along BMU
    public double bmv;//real random number along BMV
    public double z;                  //random normal number
//-------------------------------------------------------------------
      public cDeonYuliBMNormalTwist32D () {}
//-------------------------------------------------------------------
      public void Start()
      { base.Start();
        d = 1.0 / ((double)base.N1 + 1.0);    //number step
      }
//-------------------------------------------------------------------
      public double Next()
      { base.Next( ref u, ref v);    //a point on plane U x V
        bmu = ((double)u + 1.0) * d;                  //in (0,1]
        bmv = ((double)v + 1.0) * d;                  //in (0,1]
        z = Math.Sqrt(-2.0 * Math.Log(bmu)) *
            Math.Cos(2.0 * Math.PI * bmv);
        return z;                     //random normal number
      }
//=====================================
  }
}
```

To verify the correct calculation of parameters of the mathematical expectation and variance using the *cDeonYuliBMNormalTwist32D* generator, let's apply code *P050301* below, which generates the random normal variables using the Box-Muller model. The random twisting plane utilizes a grid for uniform random whole variables of length $w = 3$. Other values for the bit length w could be specified directly in the program. The total amount of the random normal variables z is $N^2 = (2^w)^2 = 2^{2w} = 2^{2 \cdot 3} = 64$. Only the calculations of the amounts of negative values kn, then zero meanings $k0$ and

after that positive values *kp* of *z* variables (17) are printed out on monitor together with mathematical expectation $Mz = E_1(Z)$ and dispersion $Dz = E_2(z) = E_1(z^2)$.

```
using nsDeonYuliBMNormalTwist32D;
                                    //twister normal numbers
namespace P050301
{ class cP050301
  { static void Main(string[] args)
    { cDeonYuliBMNormalTwist32D GN =
                 new cDeonYuliBMNormalTwist32D();
      int w = 3;                  //bit length of integer number
      GN.SetW(w);                                 //set bit length
      GN.Start();                                //generator starts
      uint N1 = GN.N1;           //maximal integer number
      uint N = N1 + 1;
      uint N2 = N * N;
      Console.WriteLine("w = {0} N = {1} N2 = {2}",
                              w, N, N2);
      int kn = 0, k0 = 0, kp = 0;
            double d = 1.0/(double)N;
      double Mz = 0.0;                            //first moment
                  double Dz = 0.0;
      for (int k = 0; k < N2; k++)
      { double z = GN.Next();      //random normal number
       if (-d < z && z <d) k0++;
       else if (z >= d) kp++; else kn++;
       Mz += z;
       Dz += z * z;
      }
      Console.WriteLine(
                               "kn = {0}   k0 = {1} kp = {2}",
                                  kn, k0, kp);
     Mz/= (double)N2;             //mathematical expectation
     Console.WriteLine("Mz = {0,12:E4}", Mz);
     Dz = Dz/(double)N2 - Mz * Mz;              //dispersion
     Console.WriteLine("Dz = {0,6:F4}", Dz);
      Console.ReadKey();                        //result viewing
     }
  }
}
```

After executing the program *P050301*, the next listing appears on the monitor:

```
w = 3 N = 8 N2 = 64
kn = 21 k0 = 22 kp = 21
Mz = -4.5653E-017
Dz = 0.7539
```

This result shows that $N2 = 64$, which means that the following random normal variables are generated: 21 positive and 21 negative numbers and then 22 zero ones. The meaning of the mathematical expectation is very close to zero, i.e. $Mz = E_1(z) = -4.5653 \cdot 10^{-17}$. With such a small bit length $w = 3$, i.e. in case of numbers 0,1,2,3,4,5,6 and 7, the dispersion of the real normal variables is $Dz = E_2\left(z - E_1(z)\right) = E_1\left(\left(z - E_1(z)\right)^2\right) = 0.7539$.

The following next program code allows tracing the convergence of moments $E_1(z)$ and $E_2(z)$ as a function of bit length *w* of the initial whole uniform random variables.

```
using nsDeonYuliBMNormalTwist32D;
                                    //twister normal numbers
namespace P050302
 { class cP050302
   { static void Main(string[] args)
     { cDeonYuliBMNormalTwist32D GN =
                      new cDeonYuliBMNormalTwist32D();
       for (int w = 3; w <= 14; w++)
       { GN.SetW(w);                              //set bit length
         GN.Start();                             //generator starts
         uint N1 = GN.N1;       //maximum integer number
         uint N = N1 + 1;
         uint N2 = N * N;
         Console.Write(
             "w = {0,2} N = {1,5} N2 = {2,9}", w, N, N2);
         double Mz = 0.0;        //mathematical expectation
         double Dz = 0.0;                          //dispersion
         for (int k = 0; k < N2; k++)
         { double z = GN.Next(); //random normal number
           Mz += z;
           Dz += z * z;
         }
          Mz /= N2;              //mathematical expectation
          Console.Write(" Mz = {0,12:E4}", Mz);
          Dz = Dz / (double)N2 - Mz * Mz;     //dispersion
          Console.WriteLine(" Dz = {0,6:F4}", Dz);
       }
       Console.ReadKey();                    //result viewing
     }
   }
}
```

After starting this program, the following strings appear:

```
w =  3   N = 8        N2 = 64          Mz = -4.5653E-017
Dz = 0.7539
w =  4   N = 16       N2 = 256         Mz = -7.0256E-017
Dz = 0.8556
w =  5   N= 32        N2 = 1024        Mz = -7.6978E-017
Dz = 0.9170
w =  6   N = 64       N2 = 4096        Mz = -5.1716E-017
Dz = 0.9531
w =  7   N = 128      N2 = 16384       Mz =  2.1413E-018
Dz = 0.9739
w =  8   N = 256      N2 = 65536       Mz = -6.2992E-017
Dz = 0.9856
w =  9   N = 512      N2 = 262144      Mz = -4.6107E-017
Dz = 0.9921
w = 10   N = 1024     N2 = 1048576     Mz = -7.3673E-017
Dz = 0.9957
```

w = 11 N = 2048 N2 = 4194304 Mz = -5.8537E-017 Dz = 0.9977
w = 12 N = 4096 N2 = 16777216 Mz = -4.5401E-017 Dz = 0.9988
w = 13 N = 8192 N2 = 67108864 Mz = -4.6954E-017 Dz = 0.9993
w =14 N =16384 N2 = 268435456 Mz = -3.6657E-017 Dz = 0.9996

These results confirm that if a grid of the twisting uniform plane is increasing then the mathematical expectation within the error of computations is very close to 0 and the dispersion tends to 1. This corresponds to the basic parameters of the standard normal distribution having $E_1(\xi) = 0$ and $E_2(\xi) = E_1(\xi^2) = 1$.

Discussion

In evaluating the transformation of stochastic processes the methods of mathematical statistics are used broadly (Wackerly *et al*., 2008). Particularly, the widespread one is Pearson's χ^2 test, which provides the possibility in an estimation of proximity between two things: the probability distribution of the observed random variables z and the probability distribution of the assumed (hypothetical) random variables ξ. The essence of this correspondence lies in the analysis of the considerably rare random variables z and ξ.

With the help of expression (17), grid nodes in uniform plane $BMU \times BMV = (0: 1] \times (0: 1]$ are mapped to the random normal variables z with the probability distribution function $F_\Xi(\xi)$ (1). To confirm this hypothesis, let's divide interval L from *min* to *max* values $L = [z_{min}, z_{max}]$ into several subintervals d_i. The length L of all subintervals d_i is understood as the difference between *max* and *min* values of the observed random variables $z \in [z_{min}, z_{max}]$. All values of d_i could be chosen arbitrarily, but usually uniform length d is applied, provided that:

$$z_{\max} - z_{\min} = L = \sum_{i=1}^{n_L} d_i = n_L \cdot d \tag{19}$$

Now, it follows from this expression (19) that:

$$d = \frac{z_{\max} - z_{min}}{n_L} \tag{20}$$

The value of n_L in (20) could be calculated from the conventional approach of binary multiplicity when estimating the sufficiency of observations:

$$n_L = \left\lfloor \log_2 N^* \right\rfloor + 1 \tag{21}$$

Expression (21) is applied when value N^* isn't a multiple of the power function 2^x of some whole variable x. In the proposed implementation of model (17), a grid on the random plane $BMU \times BMV$ is used, which contains the complete uniform sequences U and V with the number of grid nodes $N^2 = \left(2^w\right)^2 = 2^{2w}$. In this case, the value of $log_2 N^2$ has no remainder. Thus, expression (21) could be reduced to the following form:

$$n_L = \log_2 N^* = \log_2 2^{2w} = 2w \tag{22}$$

On each subinterval $d_{i \in \overline{[1,2w]}} = \left[z_{\min} + (i-1) \cdot d, z_{\min} + i \cdot d \right]$, there are the random normal variables in amount v_i. The observed probability $g(z \in d_i)$ is given by the ratio of the total number $N^2 = 2^{2w}$ of observations:

$$g\left(z \in d_i\right) = \frac{v_i}{N^2} \tag{23}$$

The values of the theoretical hypothetical probabilities $h(\xi \in d_i)$ on the same intervals are calculated as the local integral values:

$$h\left(\xi \in d_i\right) = \int_{z_{\min}+(i-1)d}^{z_{\min}+i \cdot d} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx \tag{24}$$

The ideal normal generator (23) has to be such that computations (23) and (24) coincide, i.e. $g\left(z \in l_i\right) = h(\xi \in l_i)$. For evaluation of such a coincidence, Pearson proposed using the test of criterion of significance (Cramer, 1999).

This significance test is based on the observation results, i.e. the differences between observed $g(z \in d_i)$ and hypothetical $h(\xi \in d_i)$ probabilities at the corresponding subintervals d_i are calculated. These differences are considered as joint random events $< g_i, h_i >$. It should be noted that the events of these probabilities are really independent, since they are formed by random independent real and hypothetical trials. Because the differences of probabilities $\delta_i = g_i = h_i$ can be either positive or negative, that suggests evaluating their square meanings $\delta_i^2 = \left(g_i - h_i\right)^2$ since they are positive only. Thus, it is possible to obtain an analytical value of χ^2 for the sum of squares of the joint random variables δ_i^2:

$$\chi^2 = \sum_{i=1}^{n_L = 2w} \delta_i^2 = \sum_{i=1}^{n_L = 2w} \left(g_i - h_i\right)^2 \tag{25}$$

Summarizing the squares of the probability differences (25) having the weighting coefficients c_j, a numerical estimation of the Pearson's criterion appears as follows:

$$Q = \sum_{i=1}^{2w} c_i \left(g_i - h_i \right)^2 \tag{26}$$

The proposal is that choice of coefficients c_i is carried out as inversely proportional to the significance of hypothetical probabilities:

$$c_i = \frac{N}{h_i} \tag{27}$$

With this selection of coefficients c_i (27) the Pearson's significance criterion (26) takes the following form:

$$Q = N \sum_{i=1}^{2w} \frac{\left(g_i - h_i \right)^2}{h_i} = \sum_{i=1}^{2w} \frac{\left(v_i - N \cdot h_i \right)^2}{N \cdot h_i} \tag{28}$$

Pearson's proof is that the criterion of significance Q (28) can be related to the distribution χ^2 of the sum of squares of the random variables:

$$F\left(\chi^2 \right) = \int_0^{\chi^2} \frac{1}{2^{L/2} \Gamma\left(L/2 \right)} x^{\frac{L}{2}-1} e^{-\frac{x}{2}} dx \tag{29}$$

Designation $\Gamma(L/2)$ in (29) is obviously a gamma-function:

$$\Gamma\left(\lambda = L/2 \right) = \int_0^{\infty} x^{\frac{L}{2}-1} e^{-x} dx \tag{30}$$

In expressions (29) and (30), parameter L is the number of degrees of freedom which is associated with the amount of subintervals n_L as follows:

$$L = n_L - m \tag{31}$$

Usually parameter m is defined as the number of superimposed links. As the first such a connection, it usually distinguishes the classical condition of probability theory:

$$\sum_{i \in I} p_i = 1 \tag{32}$$

If this condition is taken into account, then m = 1 and = n_L - 1. It is forced to do so if nothing is known about the completeness of the observations. To superimposed links also often include the coincidence of mathematical expectations and the coincidence of variances. In the last case m = 3, although this is clearly not obvious and, possibly, doubtful. However, generator *cDeonYuliBMNormalTwist32D* proposed above generates the normal values for the complete sequences of the twisting whole plane, i.e. all the arguments about superimposed links are already fully taken into account in the used generation of the complete twisting plane. Therefore, there is no need to adjust (31), i.e. using m = 0 and $L = n_L$.

So, implementation of the significance criterion χ^2 consists in using the following expression:

$$1 - \alpha = F\left(\chi_\alpha^2 \right) = \int_0^{\chi_\alpha^2} \frac{1}{2^{n_L/2} \Gamma\left(L/2 \right)} x^{\frac{n_L}{2}-1} e^{-\frac{x}{2}} dx \tag{33}$$

Solving this integral expression with respect to χ_α^2, a value which is possible to compare with the criterion of significance Q (28) is obtained. The significance level, i.e. parameter α is usually chosen as 0.05 or 0.1 or 0.15. If $Q \le \chi_\alpha^2$, then the hypothesis of correspondence between the investigated distributions is accepted. This means that the observed random values correspond to a given hypothetical distribution with a probability of at least 1 -α, i.e. hypothetical extra-large deviations could be ignored. This assumption is usually made by researchers when they determine the insignificance of the level of possible meaningful deviations. If the significance level is assumed to be α = 0.05, then the observed statistics have a given distribution with a probability not worse than 1-α = 1– 0.05=0.95.

Below is the program code, which verifies the result of testing *cDeonYuliBMNormalTwist32D* generator for compliance with the standard normal distribution of the received random variables. The calculation of the exact integrals is performed absolutely accurate according to a given error by using Darboux-Riemann technique in function *DarbouxRiemann()*.

```
using nsDeonYuliBMNormalTwist32D;
                                    //normal number generator
namespace P050401
 { delegate double delF(double x);
//------------------------------------------------------------------
  class cP050401
  { static void Main(string[] args)
    { cDeonYuliBMNormalTwist32D GN =
                      new cDeonYuliBMNormalTwist32D();
      int w = 13;   //bit length of integer random numbers
      GN.SetW(w);                              //set bit length
      GN.Start();                             //generator starts
      uint N1 = GN.N1;           //maximal integer number
      uint N = N1 + 1;             //twister sequence length
      uint N2 = N * N;     //point quantity on twister plane
      Console.WriteLine("w = {0} N = {1} N2 = {2}",
                         w, N, N2);
      int nL = 2 * w;          //interval quantity in hi-square
      Console.WriteLine("nL = {0}", nL);
      double zmin = 0.0, zmax = 0.0;
      double[] z = new double[N2];       //normal numbers
```

DOI: 10.3844/jcssp.2020.1.13

```
 double MZ = 0.0;                    //first moment E1
 double DZ = 0.0;                   //second moment E2
 for (int k = 0; k < N2; k++)       //all points on plane
 { double zz = GN.Next();   //random normal number
  if (zz < zmin) zmin = zz;
  else if (zmax < zz) zmax = zz;
  z[k] = zz;                        //random normal number
  MZ += zz;                  //mathematical expectation
   DZ += zz * zz;                              //dispersion
}
Console.WriteLine(
"zmin = {0,7:F4} zmax = {1,7:F4}", zmin, zmax);
double d = (zmax - zmin)/nL;          //interval length
Console.WriteLine("d = {0:F4}", d);
 int[] gnu = new int[nL];      //quantity under intervals
 for (int i = 0; i < nL; i++) gnu[i] = 0;
 for (int k = 0; k < N2; k++)
    for (int i = 1; i <= nL; i++)
       if (((zmin +(i-1) * d ) <= z[k]) &&
           (z[k] <= (zmin + i*d)))
       { gnu[i - 1]++;
         break;
       }
 MZ/= N2;                    //mathematical expectation
 Console.WriteLine("MZ = {0:E5}", MZ);
DZ = DZ/(double)N2 - MZ * MZ;          //dispersion
 Console.WriteLine("DZ = {0:F5}", DZ);
 double[] g = new double[nL];   //viewing probability
 double[] h = new double[nL];    //hypoth. probability
 int[] hnu = new int[nL];    //hypothetical frequencies
 double sh = 0.0;         //hypothetical probability sum
 int sgnu = 0;     //random number quantity in interval
 Console.Write(" i gnu hnu h");
 Console.WriteLine(" a b");
 for (int i = 0; i < nL; i++)
{ g[i] = (double)gnu[i]/(double)N2;
 double a = zmin + i * d;              //interval left edge
 double b = a + d;                    //interval right edge
 h[i] = DarbouxRiemann(fx, a, b, 0.001);     //integral
 hnu[i] = (int)(h[i] * N2);          //quantity of numbers
 sh += h[i];                //hypothetical probability sum
 sgnu += gnu[i];                          //all intervals sum
 Console.Write("{0,2} {1,7} {2,7} {3,8:F5}",
                    i, gnu[i], hnu[i], h[i]);
 Console.WriteLine(" {0,8:F4} {1,8:F4}", a, b);
}
Console.WriteLine("sgnu = {0}", sgnu);
Console.WriteLine("sh = {0:F5}", sh);
double Q = 0.0;       //Pearson's significance criterion
for (int i = 0; i < nL; i++)
 { double dp = g[i] - h[i];
   Q += dp * dp/h[i];
}
Q *= (double)N2;     //Pearson's significance criterion
Console.WriteLine("Q = {0:F5}", Q);
   Console.ReadKey();                    //result viewing
  }
//---------------------------------------------------------------------
  static double fx(double x)
   { return Math.Exp(-x * x / 2.0)/
            Math.Sqrt(2.0 * Math.PI);
    }
//---------------------------------------------------------------------
  static double DarbouxRiemann(delF f, double a,
                                  double b, double e)
  { double f1 = 0.0;          //function on left square edge
    double f2 = 0.0;         //function on right square edge
    double S1 = 0.0;                 //Darboux lower sum
    double S2 = 0.0;                 //Darboux upper sum
     double dx = (b - a)/100.0;
     do
     { S1 = 0.0; S2 = 0.0;           //initial value of sums
     double dxR = dx;
     for (double x = a; x < b - dx/2.0; x += dx)
     { f1 = f(x);                              //left edge value
       if (x > b - 1.4 * dx) dxR = b - x;                //last
       f2 = f(x + dxR);                    //right edge value
       if (f1 <= f2)
       { S1 += f1 * dx;                 //Darboux lower sum
         S2 += f2 * dx;                 //Darboux upper sum
       }
      else                                   //descending area
      { S1 += f2 * dx;                  //Darboux lower sum
        S2 += f1 * dx;                  //Darboux upper sum
      }
     }
     dx/= 2.0;                       //reduce the area by half
     } while (Math.Abs(S2 - S1) > e);  //Riemann cond-n
     return (S1 + S2) / 2.0;           //value in the middle
    }
//=====================================
  }
}
```

After running the program *P050401*, the following listing appears on the monitor:

```
w = 13 N = 8192 N2 = 67108864
nL = 26
zmin = -4.2452 zmax = 4.2452
d = 0.3266
MZ = -4.6954E-17
DZ = 0.99934
i     gnu       hnu       h         a         b
0     2213      2255      0.00003   -4.2452   -3.9187
1     7853      8018      0.00012   -3.9187   -3.5921
2     25565     25652     0.00038   -3.5921   -3.2655
3     73708     73829     0.00110   -3.2655   -2.9390
4     191021    191163    0.00285   -2.9390   -2.6124
5     445232    445308    0.00664   -2.6124   -2.2859
6     933149    933259    0.01391   -2.2859   -1.9593
```

7	1759559	1759673	0.02622	-1.9593	-1.6328
8	2984954	2985055	0.04448	1.63280	-1.3062
9	4555737	4555813	0.06789	-1.3062	-0.9797
10	6255547	6255678	0.09322	-0.9797	-0.6531
11	7728140	7728218	0.11516	-0.6531	-0.3266
12	8595850	8589771	0.12800	-0.3266	0.00000
13	8587658	8589771	0.12800	0.00000	0.32660
14	7728140	7728218	0.11516	0.32660	0.65310
15	6255547	6255678	0.09322	0.65310	0.97970
16	4555737	4555813	0.06789	0.97970	1.30620
17	2984954	2985055	0.04448	1.30620	1.63280
18	1759559	1759673	0.02622	1.63280	1.95930
19	933149	933259	0.01391	1.95930	2.28590
20	445232	445308	0.00664	2.28590	2.61240
21	191021	191163	0.00285	2.61240	2.93900
22	73708	73829	0.00110	2.93900	3.26550
23	25565	25652	0.00038	3.26550	3.59210
24	7853	8018	0.00012	3.59210	3.91870
25	2213	2255	0.00003	3.91870	4.24520

sgnu = 67108864
sh = 0.99998
Q = 14.54771

This listing shows that the tests are carried out on the twisting plane of whole random variables having w = 13 bits length. The plane contains a grid with the amount of $N2 = N^2 = \left(2^w\right)^2 = 2^{2w} = 2^{26} = 67108864$ nodes. So, in program *P050401* a total of 2^{2w} = 67108864 random normal values are generated. The minimum meaning of values *zmin* is $z_{\min}$ = -4.2452; the maximum one *zmax* turned out to be symmetrical is $z_{\max}$ = 4.2452. According to expression (22), it is recommended using $n_L = 2w = 26$ subintervals from $z_{\min}$ = -4.2452 to $z_{\max}$ = 4.2452. By using (22) it follows that all the subintervals have the same length $d = \left(z_{\max} - z_{\min}\right)/n_L = 8.4904/26 = 0.3266$. The mathematical expectation of the created normal values is very close to 0 and it is $MZ = E1(z) = -4.6954 \cdot 10^{-17}$. Dispersion is 0.99934 and it approaches to 1. Then in this listing there are lines of all the subintervals, each of them indicates quantity *gnu* of the generated normal values in this subinterval; then follows quantity *hnu* of the hypothetical random variables; after that appears the hypothetical probability of subinterval (24); at last, this listing closes beginning *a* and ending *b* of this subinterval. Controlling the strict correspondence of the total amount of generations $N2 = N^2$ = 67108864 and generation by the intervals provides the total number *sgmu* = 67108864 of all the random variables in subintervals. These values are the same and they are multiples of the distribution over the subintervals without a remainder, keeping the degree of freedom equal to the number of subintervals $L = n_L = \log_2 2^{2w} = 26$. The sum of the hypothetical interval probabilities is *sh* = 0.99998, which confirms the basic identity of probability theory. The Pearson's significance criterion according to (28) is Q = 14.54771.

To draw conclusions using criterion Q regarding the correspondence of the generated random variables to the normal standard distribution (1), it is necessary to calculate χ^2_α for a given significance level α in accordance to expression (33). It uses gamma-function (30), for which one of the important properties is its factorial character:

$$\Gamma(\lambda) = \int_0^\infty x^{\lambda-1} e^{-x} dx = (\lambda - 1)! \tag{34}$$

If λ is an integer-numbered, then there is no need to calculate this integral directly. In the tests of program *P050401*, it is used gamma-function $\Gamma(\lambda = n_L/2) = \Gamma(26/2) = (13-1)! = 479001600$. Taking into account the factorial properties of gamma-function, the transformation of integral (29) has the following form:

$$F\left(\chi^2, nL = 26\right) = \int_0^{\chi^2} \frac{1}{2^{n_L/2}\Gamma(n_L/2)} x^{\frac{n_L}{2}-1} e^{-\frac{x}{2}} dx = \int_0^{\chi^2} \frac{1}{2^{13}\Gamma(13)} x^{12} e^{-\frac{x}{2}} dx = \\ = \int_0^{\chi^2} \frac{1}{2^{13} \cdot 12!} x^{12} e^{-\frac{x}{2}} dx = \int_0^{\chi^2} \frac{1}{8192 \cdot 479001600} x^{12} e^{-\frac{x}{2}} dx \tag{35}$$

With the significance level α = 0.05, the last integral in (35) has to ensure the following:

$$F\left(\chi^2_\alpha, r = 26, \alpha = 0.05\right) = \int_0^{\chi^2_\alpha} \frac{1}{8192 \cdot 479001600} x^{12} e^{-\frac{x}{2}} dx = \\ = 1 - \alpha = 0.95 \tag{36}$$

This expression is satisfied with an accuracy of 0.00001 for $\chi^2_\alpha = 38.8859$. Comparing value χ^2_α with Pearson's significance criterion Q obtained earlier in program *P050401*, it is obvious that $Q = 14.5477 < \chi^2_\alpha = 38.8859$. It means that with a probability of at least 0.95, generator *cDeonYuliBMNormalTwist32D* creates indeed the random values with a standard normal distribution according to the Box-Muller model, having the mathematical expectation $M = E_1(z) = 0$ and dispersion $D = E_2(z) = E_1(z^2) = 1$.

Conclusion

Analysis of the source materials shows that algorithms of the modern generators used for the random normal variables do not have a sufficient evaluation level according to the statistical criterion χ^2 of significance. In order to increase the

approximation of the generated random normal variables to the standard normal distribution with zero mathematical expectation and singular dispersion, we propose the use of generators with the model of Box-Muller transformation based on a technique of absolutely complete uniform twisting planes. However, their direct application is limited by the required properties of half-open single Descartes uniform planes. This is taken into account in designed special class *nsDeonYuliBMNormalTwist32D*, which combines the technology of twisting half-open Descartes planes and Box-Muller transformation algorithm. The experiments confirm undoubtedly the notable level 1-α of Pearson's criterion of significance χ^2_α. Moreover, the variety of initial twisting planes provides a possibility obtaining many different generations of twisters for each pair of congruential constants. In the future, the obtained results of this work could be implemented in a large number of applied tasks which use the random normal numbers.

Acknowledgment

The authors are thankful to Matthew Vandenberg, Jacqueline Nolan, J. Alex Watts and Walter Harrington (University of Arkansas for Medical Sciences, Little Rock, AR, USA) for the proofreading.

Funding Information

The authors have no support or funding to report.

Author's Contributions

Both authors equally contributed to this work.

Ethics

This article is original and contains unpublished material. The corresponding author confirms that all of the other authors have read and approved the manuscript. No ethical issues were involved and the authors have no conflict of interest to disclose.

References

Box, G.E.P. and M.E. Muller, 1958. A note on the generation of random normal deviates. Annals Math. Stat., 29: 610-611. DOI: 10.1214/aoms/1177706645

Cramer, H., 1999. Mathematical Methods of Statistics. 1st Edn., Princeton University Press, ISBN-10: 0691005478, pp: 575.

Deon, A. and Y. Menyaev, 2016a. The complete set simulation of stochastic sequences without repeated and skipped elements. J. Univ. Comput. Sci., 22: 1023-1047. DOI: 10.3217/jucs-022-08-1023

Deon, A. and Y. Menyaev, 2016b. Parametrical tuning of twisting generators. J. Comput. Sci., 12: 363-378. DOI: 10.3844/jcssp.2016.363.378

Deon, A. and Y. Menyaev, 2017. Twister generator of arbitrary uniform sequences. J. Univ. Comput. Sci., 23: 353-384. DOI: 10.3217/jucs-023-04-0353

Deon, A. and Y. Menyaev, 2018. Uniform twister plane generator. J. Comput. Sci., 14: 260-272. DOI: 10.3844/jcssp.2018.260.272

Deon, A. and Y. Menyaev, 2019. Poisson twister generator by cumulative frequency technology. Algorithms, 12: 114-118. DOI: 10.3390/a12060114

Feller, W., 2008. An Introduction to Probability Theory and its Applications. 3rd Edn., WSE Press, ISBN-10: 8126518057, pp: 528.

Gnedenko, B., 1998. Theory of Probability. 6th Edn., CRC Press, ISBN-10: 9056995855, pp: 520.

Halim, Z.A., N.C. How and S.J.J. Ong, 2012. FPGA based RNG for random WOB method in unit cube capacitance calculation. Proceedings of the IEEE Asia-Pacific Conference on Applied Electromagnetics, Dec. 11-13, IEEE Xplore Press, Melaka, Malaysia, pp: 11-16. DOI: 10.1109/APACE.2012.6457622

Kolmogorov, A.N. and S.V. Fomin, 1999. Elements of the Theory of Functions and Functional Analysis. 1st Edn., Mineola NY, ISBN-10: 0486406830, pp: 128.

Kolmogorov, A.N., 1968. Three approaches to the quantitative definition of information. Int. J. Comput. Math., 2: 157-168. DOI: 10.1080/00207166808803030

Lee, D.U., J.D. Villasenor, W. Luk and P.H.W. Leong, 2006. A hardware gaussian noise generator using the box-muller method and its error analysis. IEEE Trans. Comput., 55: 659-671. DOI: 10.1109/TC.2006.81

Malik, J.S., J.N. Malik, A. Hemani and N.D. Gohar, 2011. An efficient hardware implementation of high quality AWGN generator using box-muller method. Proceedings of the 11th International Symposium on Communications and Information Technologies, Oct. 12-14, IEEE Xplore Press, Hangzhou, China, pp: 449-454. DOI: 10.1109/ISCIT.2011.6090035

Martino, L., D. Luengo and J. Miguez, 2012. Efficient sampling from truncated bivariate Gaussians via box-muller transformation. Electron. Lett., 48: 1533-34. DOI: 10.1049/el.2012.2816

Menyaev, Y.A. and V.P. Zharov, 2005. Phototherapeutic technologies for oncology. Proc. SPIE, 5973: 271-278. DOI: 10.1117/12.640217

Menyaev, Y.A. and V.P. Zharov, 2006a. Experience in development of therapeutic photomatrix equipment. Biomed. Eng., 40: 57-63. DOI: 10.1007/s10527-006-0042-6

Menyaev, Y.A. and V.P. Zharov, 2006b. Experience in the use of therapeutic photomatrix equipment. Biomed. Eng., 40: 144-147. DOI: 10.1007/s10527-006-0064-0

Menyaev, Y.A., D.A. Nedosekin, M. Sarimollaoglu, M.A. Juratli and E.I. Galanzha, *et al*., 2013. Optical clearing in photoacoustic flow cytometry. Biomed. Opt. Express, 4: 3030-41. DOI: 10.1364/BOE.4.003030

Menyaev, Y.A., K.A. Carey, D.A. Nedosekin, M. Sarimollaoglu and E.I. Galanzha *et al*., 2016. Preclinical photoacoustic models: Application for ultrasensitive single cell malaria diagnosis in large vein and artery. Biomed. Opt. Express, 7: 3643-58. DOI: 10.1364/BOE.7.003643

Neugebauer, F., I. Polian and J.P. Hayes, 2019. On the limits of stochastic computing. Proceedings of the IEEE International Conference on Rebooting Computing, Nov. 6-8, IEEE Xplore Press, San Mateo, CA, USA, pp: 1-8. DOI: 10.1109/ICRC.2019.8914706

Paraskevakos, I. and V. Paliouras, 2011. A flexible high-throughput hardware architecture for a gaussian noise generator. Proceeding of the International Conference on Acoustics, Speech and Signal Processing, May 22-27, IEEE Xplore Press, Prague Czech Republic, pp: 1673-1676. DOI: 10.1109/ICASSP.2011.5946821

Sauer, T., 2012. Numerical solution of stochastic differential equations in finance. Handbook Computat. Finance Chapter, 19: 529-550. DOI: 10.1007/978-3-642-17254-0_19

Sukajaya, I.N., A.V. Vitianingsih, S.N.S. Mardi, K.E. Purnama and M. Hariadi *et al*., 2012. Multi-parameter dynamic difficulty game's scenario using box-muller of gaussian distribution. Proceedings of the 7th International Conference on Computer Science and Education, Jul. 14-17, Melbourne VIC Australia, pp: 1666-1671. DOI: 10.1109/ICCSE.2012.6295384

Wackerly, D., W. Mendenhall and R.L. Scheaffer, 2008. Mathematical Statistics with Applications. 7th Edn., Thompson Brooks/Cole, ISBN-10: 0495110817, pp: 944.

Wikipedia. Box-Muller transform. https://en.wikipedia.org/wiki/Box-Muller_transform

Zhou, Y., N. Wang and X. Jiang, 2014. Parallel gaussian white noise generator based on cellular automaton theory and box muller algorithm. Proceedings of the International Conference on Wireless Communication and Sensor Network, Dec. 13-14, IEEE Xplore Press, Wuhan China, pp: 143-147. DOI: 10.1109/WCSN.2014.36