

The Stochastic Processes Generation in OpenModelica

M. N. Gevorkyan,^{1,*} A. V. Demidova,^{1,†} A. V. Korolkova,^{1,‡} D. S. Kulyabov,^{1,2,§} and L. A. Sevastianov^{1,3,¶}

¹*Department of Applied Probability and Informatics,
Peoples' Friendship University of Russia (RUDN University),
6 Miklukho-Maklaya str., Moscow, 117198, Russia*

²*Laboratory of Information Technologies
Joint Institute for Nuclear Research
6 Joliot-Curie, Dubna, Moscow region, 141980, Russia*

³*Bogoliubov Laboratory of Theoretical Physics
Joint Institute for Nuclear Research
6 Joliot-Curie, Dubna, Moscow region, 141980, Russia*

Background: Component-based modeling language Modelica (OpenModelica is open source implementation) is used for the numerical simulation of complex processes of different nature represented by ODE system. However, in OpenModelica standard library there is no routines for pseudo-random numbers generation, which makes it impossible to use for stochastic modeling processes.

Purpose: The goal of this article is a brief overview of a number of algorithms for generation a sequence of uniformly distributed pseudo random numbers and quality assessment of the sequence given by them, as well as the ways to implement some of these algorithms in OpenModelica system.

Methods: All the algorithms are implemented in C language, and the results of their work tested using open source package DieHarder. For those algorithms that do not use bit operations, we describe there realisation using OpwnModelica. The other algorithms can be called in OpenModelica as C functions

Results: We have implemented and tested about nine algorithms. DieHarder testing revealed the highest quality pseudo-random number generators. Also we have reviewed libraries Noise and AdvancedNoise, who claim to be adding to the Modelica Standard Library.

Conclusions: In OpenModelica system can be implemented generators of uniformly distributed pseudo-random numbers, which is the first step towards to make OpenModelica suitable for simulation of stochastic processes.

Keywords: Modelica, OpenModelica, random generator, Wiener process, Poisson process, SDE

* gevorkyan_mn@rudn.university

† demidova_av@rudn.university

‡ korolkova_av@rudn.university

§ kulyabov_ds@rudn.university

¶ sevastianov_la@rudn.university

I. INTRODUCTION

In this article we study the problem of generation uniformly distributed pseudo-random numbers, stochastic Wiener and Poisson processes in OpenModelica framework [1]. OpenModelica is one of the open source implementation of Modelica [2] modeling language (for other implementations see [3–8]). This language is designed for modeling various systems and processes that can be represented as a system of algebraic or differential equations. For the numerical solution of the equations OpenModelica uses a number of open source libraries [9–12]. However, in OpenModelica standard library there is no any function even for generating uniformly distributed pseudo-random numbers.

The first part of the article provides an overview of some algorithms for generation pseudo-random numbers, including description of pseudo-device `/dev/random` of Unix OS. For most of them we provide the algorithm written in pseudocode. We implement all described algorithms in the language of C and partly in OpenModelica. Also we tested them with dieharder — a random number generator testing suite [13].

In the second part of the paper we describe algorithms for generating normal and Poisson distributions. These algorithms are based on the generators of uniformly distributed pseudo-random numbers. Then we study the problem of computer generation of stochastic Wiener and Poisson processes.

The third part of the article has a practical focus and is devoted to the description of calling external functions, written in C language, directly from OpenModelica programs code.

II. ALGORITHMS FOR GENERATING UNIFORMLY DISTRIBUTED PSEUDO-RANDOM NUMBERS

In this section we will describe some of the most common generators of uniformly distributed pseudo-random numbers. These generators are the basis for obtaining a sequence of pseudo-random numbers of other distributions.

A. Linear congruential generator

A linear congruential generator (LCG) was first proposed in 1949 by D. H. Lehmer [14]. The algorithm II.1 is given by formula:

$$x_{n+1} = (ax_n + c) \mod m, \quad n \geq 0,$$

where m is *the mask* or *the modulus* $m > 1$, a is *the multiplier* ($0 \leq a < m$), c is *the increment* ($0 \leq c < m$), x_0 is *the seed* or initial value. The result of the repeated application of this recurrence formula is *linear congruential sequence* x_1, \dots, x_n . A special case $c = 0$ is called *multiplicative* congruential method.

Algorithm II.1 LCG — linear congruential generator

Require: n , $seed$
const $m \leftarrow 2^{64}$
const $a \leftarrow 6364136223846793005$
const $c \leftarrow 1442695040888963407$
 $x_0 \leftarrow seed$
for $i = 0$ to n **do**
 $x_i = (a \cdot x_{i-1} + c) \mod m$
end for
return $\{x_0, x_1, \dots, x_n\}$

The number m , a , c is called a «magic» or «magic» because their values are specified in the code of the program and are selected based on the experience of the use of the generator. The quality of the generated sequence depends essentially on the correct choice of these parameters. The sequence $\{x\}_1^n$ is periodic and its period depends on the number m , which must therefore be large. In practice, one chooses m equal to the machine word size (for 32-bit architecture — 2^{32} , for 64-bit architecture — 2^{64}). D. Knuth [14] recommends to choose

$$a = 6364136223846793005, \quad c = 1442695040888963407, \quad m = 2^{64} = 18446744073709551616.$$

In the article [15], you can find large tables with optimal values a , b and m .

Also there are generalisations of LCG, such as quadratic congruential method $x_n = (ax_{n-1}^2 + bx_{n-1} + d) \mod m$ cubic congruential method $x_n = (ax_{n-1}^3 + bx_{n-1}^2 + cx_{n-1} + d) \mod 2^e$.

Currently, the linear congruential method has mostly a historical value, as it generates relatively low-quality pseudo-random sequence compared to other, equally simple generators.

B. Lagged Fibonacci generator

The lagged Fibonacci generation can be considered as the generalization of the linear congruential generator. The main idea of this generalisation is to use multiple previous elements to generate current one. Knuth [14] claims that the first such generator was proposed in the early 50-ies and based on the formula:

$$x_{n+1} = (x_n + x_{n-1}) \mod m.$$

In practice, however, he showed himself not the best way. In 1958 George. J. Mitchell and D. Ph. Moore invented a much better generator II.2

$$x_n = (x_{n-n_a} + x_{n-n_b}) \mod m, n \geq \max(n_a, n_b).$$

It was the generator that we now call **LFG** — **lagged Fibonacci Generator**.

Algorithm II.2 LFG — Lagged Fibonacci generator

```

 $n_a \leftarrow 55$ 
 $n_b \leftarrow 24$ 
Require:  $s_0, s_1, \dots, s_{n_b}, n \geq 0$ 
 $x_0, x_1, \dots, x_{n_b} \leftarrow r_0, r_1, \dots, r_{n_b}$ 
for  $i = (n_a + 1)$  to  $n$  do
  if  $x_{i-n_a} \geq x_{i-n_b}$  then
     $x_i = x_{i-n_a} - x_{i-n_b}$ 
  else if  $x_{i-n_a} < x_{i-n_b}$  then
     $x_i = x_{i-n_a} - x_{i-n_b} + 1$ 
  end if
end for
return  $\{x_0, x_1, \dots, x_n\}$ 

```

As in the case of LCG generator «magical numbers» n_a and n_b greatly affects the quality of the generated sequence. The authors proposed to use the following magic numbers n_a and n_b

$$n_a = 24, n_b = 55.$$

Knuth [14] gives a number of other values, starting from (37, 100) and finishing with (9739, 23209). Period length of this generator is exactly equal to $2^{e-1}(2^{55} - 1)$ when choosing $m = 2^e$.

As can be seen from the algorithm for the initialization of this generator must be used one an initial value and a sequence of $\max(n_a, n_b)$ random numbers.

In open source GNU Scientific Library (GSL) [16] *composite multi-recursive* generator are used. It was proposed in paper [17]. This generator is a generalisation of LFG may be expressed by the following formulas:

$$\begin{aligned}
 x_n &= (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) \mod m_1, \\
 y_n &= (b_1 y_{n-1} + b_2 y_{n-2} + b_3 y_{n-3}) \mod m_2, \\
 z_n &= (x_n - y_n) \mod m_1.
 \end{aligned}$$

The composite nature of this algorithm allows to obtain a large period equal to $10^{56} \approx 2^{185}$. The GSL uses the following parameter values a_i, b_i, m_1, m_2 :

$$\begin{aligned}
 a_1 &= 0, & b_1 &= 86098, & m_1 &= 2^{32} - 1 = 2147483647, \\
 a_2 &= 63308, & b_2 &= 0, & m_2 &= 2145483479, \\
 a_3 &= -183326, & b_3 &= -539608.
 \end{aligned}$$

Another method suggested in the paper [18] is also a kind of Fibonacci generator and is determined by the formula:

$$x_n = (a_1 x_{n-1} + a_5 x_{n-5}) \mod 5,$$

The GSL used the following values: $a_1 = 107374182, a_2 = 0, a_3 = 0, a_4 = 0, a_5 = 104480, m = 2^{31} - 1 = 2147483647$. The period of this generator is equal to 10^{46} .

C. Inversive congruential generator

Inverse congruential method based on the use of inverse modulo of a number.

$$x_{i+1} = (ax_i^{-1} + b) \mod m$$

where a is *multiplier* ($0 \leq a < n$), b is *increment* ($0 \leq b < n$), x_0 is initial value (seed). In addition $\text{GCD}(x_0, m) = 1$ and $\text{HCF}(a, m) = 1$ is required.

This generator is superior to the usual linear method, however, is more complicated algorithmically, since it is necessary to find the inverse modulo integers which leads to performance reduction. To compute the inverse of the number usually applies the extended Euclidean algorithm [14, §4.3.2].

D. Generators with bitwise operations

Most generators that produce high quality pseudo-random numbers sequence use bitwise operations, such as conjunction, disjunction, negation, exclusive disjunction (xor) and bitwise right/left shifting.

1. Mersenne twister

Mersenne twister considered one of the best pseudo-random generators. It was developed in 1997 by Matsumoto and Nishimura [19]. There are 32-, 64-, 128-bit versions of the Mersenne twister. The name of the algorithm derives from the use of Mersenne primes $2^{19937} - 1$. Depending on the implementation the period of this generator can be up to $2^{216091} - 1$.

The main disadvantage of the algorithm is the relative complexity and, consequently, relatively slow performance. Otherwise, this generator provides high-quality pseudo-random sequence. An important advantage is the requirement of only one initiating number (seed). Mersenne twister is used in many standard libraries, for example in the Python 3 module `random` [20].

Due to the complexity of the algorithm, we do not give its pseudocode in this article, however, the standard implementation of the algorithm created by Matsumoto and Nishimura freely available at the link <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt64.html>.

2. XorShift generator

Some simple generators (algorithms II.3 and II.4), giving a high quality pseudo-random sequence were developed in 2003 by George. Marsala (G. Marsaglia) [21, 22].

Algorithm II.3 xorshift*	Algorithm II.4 xorshift+
Require: $n, seed$ $x \leftarrow seed$ $y_0 \leftarrow x$ for $i = 1$ to n do $x \leftarrow x \oplus x \gg 12$ $x \leftarrow x \oplus x \ll 25$ $x \leftarrow x \oplus x \gg 27$ $y_i \leftarrow x \cdot 2685821657736338717$ end for return $\{y_0, y_1, \dots, y_n\}$	Require: $n, seed_1, seed_2$ for $i = 1$ to n do $x \leftarrow seed_1$ $y \leftarrow seed_2$ $seed_1 \leftarrow y$ $x = x \oplus (x \ll 23)$ $seed_2 = x \oplus y \oplus (x \gg 17) \oplus (y \gg 26)$ $z_i \leftarrow seed_2 + y$ end for return $\{z_1, \dots, z_n\}$

3. KISS generator

Another group of generators (algorithms II.5 and II.6), giving a high quality sequence of pseudo-random numbers is KISS generators family [23] (Keep It Simple Stupid). They are used in the procedure `random_number()` of Fortran language (`gfortran` compiler [24])

Algorithm II.5 KISS

Require: $n, seed_0, seed_1, seed_2, seed_3$
 t
for $i = 1$ **to** n **do**
 $seed_0 \leftarrow 69069 \cdot seed_0 + 123456$
 $seed_1 \leftarrow seed_1 \oplus (seed_1 \ll 13)$
 $seed_1 \leftarrow seed_1 \oplus (seed_1 \gg 17)$
 $seed_1 \leftarrow seed_1 \oplus (seed_1 \ll 5)$
 $t \leftarrow 698769069 \cdot seed_2 + seed_3$
 $seed_3 \leftarrow (t \gg 32)$
 $seed_1 \leftarrow t$
 $x_i \leftarrow seed_0 + seed_1 + seed_2$
end for
return $\{x_1, \dots, x_n\}$

Algorithm II.6 jKISS

Require: $n, seed_0, seed_1, seed_2, seed_3$
 t
for $i = 1$ **to** n **do**
 $seed_0 \leftarrow 314527869 \cdot seed_0 + 1234567$
 $seed_1 \leftarrow seed_1 \oplus (seed_1 \ll 5)$
 $seed_1 \leftarrow seed_1 \oplus (seed_1 \gg 7)$
 $seed_1 \leftarrow seed_1 \oplus (seed_1 \ll 22)$
 $t \leftarrow 4294584393 \cdot seed_2 + seed_3$
 $seed_3 \leftarrow (t \gg 32)$
 $seed_1 \leftarrow t$
 $x_i \leftarrow seed_0 + seed_1 + seed_2$
end for
return $\{x_1, \dots, x_n\}$

E. Pseudo devices `/dev/random` and `/dev/urandom`

<https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/drivers/char/random.c?id=refs/tags/v3.15.6#n52>

To create a truly random sequence of numbers using a computer, some Unix systems (in particular GNU/Linux) uses the collection of «background noise» from the operating system environment and hardware. Source of this random noise are moments of time between keystrokes (inter-keyboard timings), various system interrupts and other events that meet two requirements: to be non-deterministic and be difficult for access and for measurement by external observer.

Randomness from these sources are added to an "entropy pool", which is mixed using a CRC-like function. When random bytes are requested by the system call, they are retrieved from the entropy pool by taking the SHA hash from it's content. Taking the hash allows not to show the internal state of the pool. Thus the content restoration by hash computing is considered to be an impossible task. Additionally, the extraction procedure reduces the content pool size to prevent hash calculation for the entire pool and to minimize the theoretical possibility of determining its content.

External interface for the entropy pool is available as symbolic pseudo-device `/dev/random`, as well as the system function:

```
void get_random_bytes(void *buf, int nbytes);
% \end{minted}
```

The device `/dev/random` can be used to obtain high-quality random number sequences, however, it returns the number of bytes equal to the size of the accumulated entropy pool, so if one needs an unlimited number of random numbers, one should use a character pseudo-device `/dev/urandom` which does not have this restriction, but it also generates good pseudo-random numbers, sufficient for the most non-cryptographic tasks.

F. Algorithms testing

A review of quality criterias of an sequence of pseudo-random numbers can be found in the third chapter of the book [14], as well as in paper [25]. All the algorithms, which we described in this articles, have been implemented in C-language and tested with Dieharder test suite, available on the official website [13].

1. Dieharder overview

Dieharder is tests suite, which is implemented as a command-line utility that allows one to test a quality of sequence of uniformly distributed pseudorandom numbers. Also Dieharder can use any generator from GSL library [16] to generate numbers or for direct testing.

- `dieharder -l` — show the list of available tests,
- `dieharder -g -1` — show the list of available random number generators; each generator has an ordinal number, which must be specified after `-g` option to activate the desired generator.

- 200 `stdin_input_raw` — to read from standard input binary stream,
- 201 `file_input_raw` — to read the file in binary format,
- 202 `file_input` — to read the file in text format,
- 500 `/dev/random` — to use a pseudo-device `/dev/random`,
- 501 `/dev/urandom` — to use a pseudo-device `/dev/urandom`.

Each pseudorandom number should be on a new line, also in the first lines of the file one must specify: type of number (`d` — integer double-precision), the number of integers in the file and the length of numbers (32 or 64 - bit). An example of such a file:

```
type: d
count: 5
numbit: 64
1343742658553450546
16329942027498366702
3111285719358198731
2966160837142136004
17179712607770735227
```

When such a file is created, you can pass it to `dieharder`

```
dieharder -a -g 202 -f file.in > file.out
```

where the flag `-a` denotes all built-in tests, and the flag `-f` specifies the file for analysis. The test results will be stored in `file.out` file.

2. Test results and conclusions

The generator	Fail	Weak	Pass
LCG	52	6	55
LCG2	51	8	54
LFG	0	2	111
ICG	0	6	107
KISS	0	3	110
jKISS	0	4	109
XorShift	0	4	109
XorShift+	0	2	111
XorShift*	0	2	111
Mersenne Twister	0	2	111
dev/urandom	0	2	111

The best generators with bitwise operations are `xorshift*`, `xorshift+` and Mersenne Twister. They all give the sequence of the same quality. The algorithm of the Mersenne Twister, however, is far more cumbersome than `xorshift*` or `xorshift+`, thus, to generate large sequences is preferable to use `xorshift*` or `xorshift+`.

Among the generators which use bitwise operations the best result showed Lagged Fibonacci generator. The test gives results at the level of `XorShift+` and Mersenne Twister. However, one has to set minimum 55 initial values to initialize this generator, thus it's usefulness is reduced to a minimum. Inverse congruential generator shows slightly worse results, but requires only one number to initiate the algorithm.

III. GENERATION OF WIENER AND POISSON PROCESSES

Let us consider the generation of normal and Poisson distributions. The choice of these two distributions is motivated by their key role in the theory of stochastic differential equations. The most General form of these equations uses two random processes: Wiener and Poisson [26]. Wiener process allows to take into account the implicit stochasticity of the simulated system, and the Poisson process — external influence.

A. Generation of the uniformly distributed pseudo-random numbers from the unit interval

Generators of pseudo-random uniformly distributed numbers are the basis for other generators. However, most of the algorithms require a random number from the unit interval $[0, 1]$, while the vast majority of generators of uniformly distributed pseudo-random numbers give a sequence from the interval $[0, m]$ where the number m depends on the algorithm and the bitness of the operating system and processor.

To obtain the numbers from the unit interval one can proceed in two ways. First, one can normalize existing pseudo-random sequence by dividing each it's element on the maximum element. This approach is guaranteed to give 1 as a random number. However, this method is bad when a sequence of pseudo-random numbers is too large to fit into memory. In this case it is better to use the second method, namely, to divide each of the generated number by m .

B. Normal distribution generation

An algorithm for normal distributed numbers generation has been proposed in 1958 by George. Bux and P. E. R. Mueller [27] and named in their honor *Box-Muller transformation*. The method is based on a simple transformation. This transformation is usually written in two formats:

- standard form (was introduce in the paper [27]),
- polar form (suggested by George Bell [28] and R. Knop [29]).

Standard form. Let x and y are two independent, uniformly distributed pseudo-random numbers from the interval $(0, 1)$, then numbers z_1 and z_2 are calculated according to the formula

$$z_1 = \cos(2\pi y)\sqrt{-2 \ln x}, \quad z_2 = \sin(2\pi y)\sqrt{-2 \ln x}$$

are independent pseudo-random numbers distributed according to a standard normal law $\mathcal{N}(0, 1)$ with expectation $\mu = 0$ and the standard deviation $\sigma = 1$.

Polar form. Let x and y — two independent, uniformly distributed pseudo-random numbers from the interval $[-1, 1]$. Let us compute additional value $s = x^2 + y^2$. If $s > 1$ or $s = 0$ then existing x and y values should be rejected and the next pair should be generated and checked. If $0 < s \leq 1$ then the numbers z_1 and z_2 are calculated according to the formula

$$z_1 = x\sqrt{\frac{-2 \ln s}{s}}, \quad z_2 = y\sqrt{\frac{-2 \ln s}{s}}$$

and are independent random numbers distributed according to a standard normal law $\mathcal{N}(0, 1)$.

For computer implementation is preferable to use a polar form, because in this case one has to calculate only single transcendental function \ln , while in standard case three transcendental functions (\ln , \sin \cos) have to be calculated. An example of the algorithm shown in figure 1

To obtain a general normal distribution from the standard normal distribution, one can use the formula $Z = \sigma \cdot z + \mu$ where $z \sim \mathcal{N}(0, 1)$, and $Z \sim \mathcal{N}(\mu, \sigma)$.

C. The generation of a Poisson distribution

To generate a Poisson distribution there is a wide variety of algorithms [30–32]. The easiest was proposed by Knut [14]. This algorithm III.1 uses uniform pseudo-random number from the interval $[0, 1]$ for it's work. The algorithm's output example is depicted on figure 2

Algorithm III.1 The generator of the Poisson distribution

Require: $seed, \lambda$

$\Lambda \leftarrow \exp(-\lambda), k \leftarrow 0, p \leftarrow 1, u \leftarrow seed$

repeat

$k \leftarrow k + 1$

$u \leftarrow rand(u)$

$p = p \cdot u$

until $p > \Lambda$

return $k - 1$

▷ generation of uniformly distributed random number

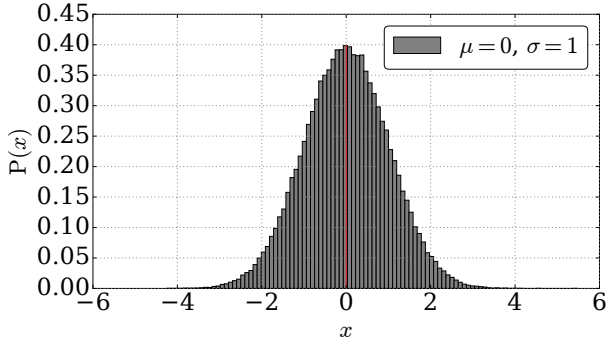


Figure 1. Normal distribution

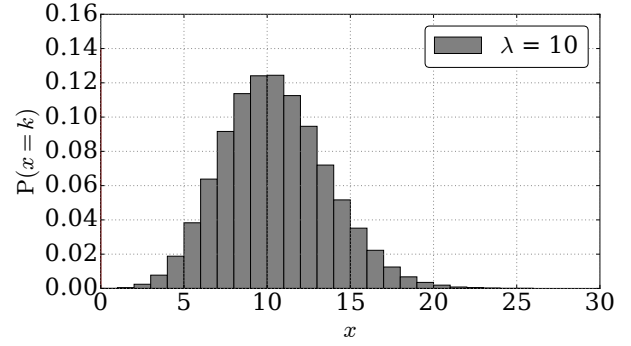


Figure 2. Poisson distribution

D. Generation of Poisson and Wiener processes

Now we going to use generators of normal and Poisson distributions to generate Wiener and Poisson stochastic processes. Let us first give the definition of these processes and then proceed to algorithms descriptions.

1. Definition of Wiener and Poisson stochastic processes

Let (Ω, \mathcal{A}, P) — probability space, where Ω — the space of elementary events \mathcal{A} — σ - algebra of subsets of Ω (a random event), P — the probability or probability measure, such that $P(\Omega) = 1$.

A family of random variables $X = \{X_t, 0 \leq t \leq T\}$, where $X_t \in \mathbb{R}^d$ will be called d -dimensional stochastic process, where a set of finite-dimensional distribution functions have the following properties (see [33, 34])

$$F_{X_{t_1}, X_{t_2}, \dots, X_{t_k}}(x_{i_1}, x_{i_2}, \dots, x_{i_k}) = P(X_{t_1} \leq x_{i_1}, X_{t_2} \leq x_{i_2}, \dots, X_{t_k} \leq x_{i_k})$$

for all $i_k = 1, 2, 3, \dots, k = 1, 2, 3, \dots, x_i \in \mathbb{R}^d$ и $t_k \in T$

The state space of X is called d -dimensional Euclidean space \mathbb{R}^d , $d = 1, 2, 3, \dots$. The time interval $[0, T]$, where $T > 0$. In numerical methods the sequence of time moments $\{t_0, t_1, t_2, \dots\}$ is used.

Random piecewise-constant process $N = \{N_t, 0 \leq t \leq T\}$ with intensity $\lambda > 0$ is called the *Poisson process* if the following properties are true (see [33, 34]):

1. $P\{N_0 = 0\} = 1$, otherwise $N_0 = 0$ almost surely.
2. N_t has independent increments: $\{\Delta N_0, \Delta N_1, \dots\}$ are independent random variables; $\Delta N_{t_i} = N_{t_{i+1}} - N_{t_i}$ and $0 \leq t_0 < t_1 < t_2 < \dots < t_n \leq T$; $\Delta N_{t_i} = N_{t_{i+1}} - N_{t_i}$ и $0 \leq t_0 < t_1 < t_2 < \dots < t_n \leq T$.
3. There is a number $\lambda > 0$ such as, for any increment ΔN_i , $i = 0, \dots, n-1$, $E[\Delta N_i] = \lambda \Delta t_i$.
4. If $P(s) = P\{N_{t+s} - N_t > 2\}$, then $\lim_{s \rightarrow 0} \frac{P(s)}{s} = 0$.

The random process $W = \{W_t, 0 \leq t \leq T\}$ is called scalar *Wiener process* (Wiener) if the following conditions are true (see [33, 34]).

1. $P\{W_0 = 0\} = 1$, otherwise $W_0 = 0$ almost surely.
2. W_t has independent increments: $\{\Delta W_0, \Delta W_1, \dots\}$ are independent random variables; $\Delta W_{t_i} = W_{t_{i+1}} - W_{t_i}$ и $0 \leq t_0 < t_1 < t_2 < \dots < t_n \leq T$.
3. $\Delta W_i = W_{t_{i+1}} - W_{t_i} \sim \mathcal{N}(0, t_{i+1} - t_i)$ где $0 \leq t_{i+1} < t_i < T$, $i = 0, 1, \dots, n-1$.

From the definition it follows that ΔW_i is normally distributed random variable with expectation $\mathbb{E}[\Delta W_i] = \mu = 0$ and variance $\mathbb{D}[\Delta W_i] = \sigma^2 = \Delta t_i$.

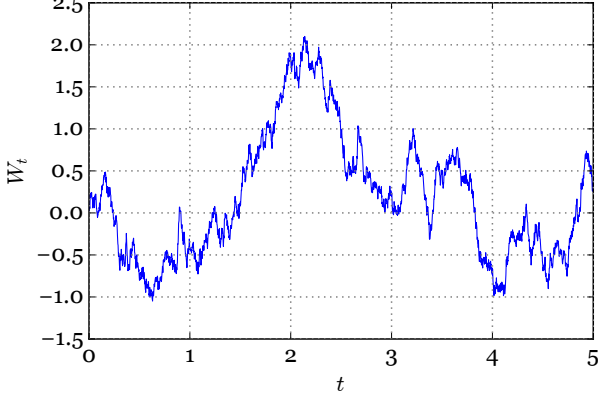


Figure 3. Wiener process

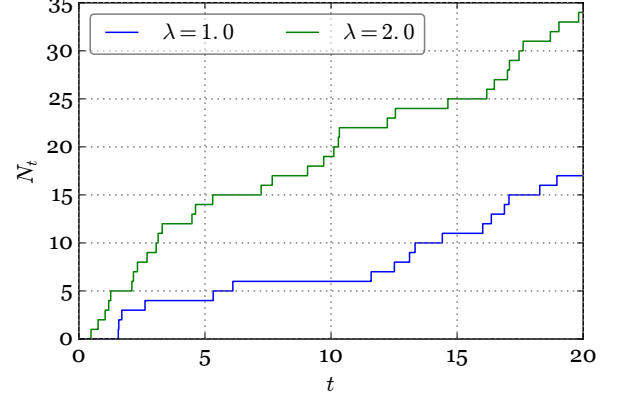


Figure 4. Poisson process

Wiener process is a model of *Brownian motion* (the random walk). The process W_t in following time points $0 = t_0 < t_1 < t_2 < \dots < t_{N-1} < t_N$ experiences random additive changes: $W_{t_1} = W_{t_0} + \Delta W_0$, $W_{t_2} = W_{t_1} + \Delta W_1$, ..., $W(t_N) = W(t_{N-1}) + \Delta W_{n-1}$, where $\Delta W_i \sim \mathcal{N}(0, \Delta t_i)$, $\forall i = 0, \dots, n-1$.

Let us write W_{t_n} in the following form: $W_{t_n} = W_{t_0} + \sum_{i=0}^{n-1} \Delta W_i$ and consider that $\mathbb{E}[\Delta W_i] = 0$ and $\mathbb{D}[\Delta W_i] = \Delta t_i$. We can show now, that the sum of normally distributed random numbers ΔW_i is also normally distributed random number:

$$\mathbb{E} \sum_{i=0}^{n-1} \Delta W_i = 0, \quad \mathbb{D} \sum_{i=0}^{n-1} \Delta W_i = \sum_{i=0}^{n-1} \Delta t_i = T, \quad \sum_{i=0}^{n-1} \Delta W_i \sim \mathcal{N}(0, T).$$

Multidimensional Wiener process $\mathbf{W}_t: \Omega \times [t_0, T] \rightarrow \mathbb{R}^m$ is defined as a random process composed of jointly independent one-dimensional Wiener processes W_t^1, \dots, W_t^m . Increments ΔW_i^α , $\forall \alpha = 1, \dots, m$ are jointly independent normal distributed random variables. On the other hand, the vector ΔW_i^α can be represented as a multidimensional normally distributed random variable with expectation vector $\mu = 1$ and a diagonal covariance matrix.

2. The generation of the Wiener process

To simulate one-dimensional Wiener process, one should generate the N normally distributed random numbers $\varepsilon_1, \dots, \varepsilon_N$ and build their cumulative sums of ε_1 , $\varepsilon_1 + \varepsilon_2$, $\varepsilon_1 + \varepsilon_2 + \varepsilon_3$. As result we will get a *trajectory* of the Wiener process $W(t)$ cm Fig. 3.

In the case of multivariate random process, one needs to generate m sequences of N normally distributed random variables.

3. The generation of a Poisson process

A simulation of the Poisson process is much like Wiener one, but now we need to generate a sequence of numbers distributed according to the Poisson law and then calculate their cumulative sum. The plot of Poisson process is shown in Fig. 4. The figure shows that the Poisson process represents an abrupt change in numbers have occurred over time events. The intensity λ depends on the average number of events over a period of time.

Because of this characteristic behavior of the Poisson process is also called an process with jumps and stochastic differential equations, with Poisson process as second driving process, are called equations with jumps [26]

IV. SIMULATION OF STOCHASTIC PROCESSES IN OPENMODELICA

As already mentioned in the introduction, there are no any pseudorandom numbers generators in OpenModelica. Thus that makes this system unusable for stochastic processes modeling. However `Noise` library `build.openmodelica`.

`org/Documentation/Noise.html` developed by Klöckner (Klöckner) [35] should be mentioned. The basis of this library are `xorshift` generators (algorithms II.3 and II.4), written in C. However, an inexperienced user may face a problem, because one needs compile C-source files first to use that library.

In this article we will describe the procedure required for connection of external C functions to OpenModelica programme. That will allow the user to install the Noise library and to connect their own random number generators. We also provide a minimal working example of stochastic Wiener process generator and the example of ordinary differential equation with additive stochastic part.

A. Connection of external C-functions to OpenModelica program

Let us consider the process of connection of external functions to modelica program. The relevant section in the official documentation misses some essential steps that's why it will lead to an error. All steps we described, had been performed on a computer with Linux Ubuntu 16.04 LTS and OpenModelica 1.11.0-dev-15.

When the code is compiled OpenModelica program is translated to C code that then is processed by C-compiler. Therefore, OpenModelica has built-in support of C-functions. In addition to the C language OpenModelica also supports Fortran (F77 only) and Python functions. However, both languages are supported indirectly, namely via wrapping them in the appropriate C-function.

The use of external C-functions may be required for various reasons, for example implementations of performance requiring components of the program, using a fullscale imperative programming language, or the use of existing sourcecode in C.

We give a simple example of calling C-functions from Modelica program. Let's create two source files: `ExternalFunc1.c` and `ExternalFunc2.c`. These files will contain simple functions that we want to use in our Modelica program.

```
// File ExternalFunc1.c
double ExternalFunc1_ext(double x)
{
    return x+2.0*x*x;
}

// File ExternalFunc2.c
double ExternalFunc2(double x)
{
    return (x-1.0)*(x+2.0);;
}
```

In the directory, where the source code of Modelica program is placed, we must create two directories: `Resources` and the `Library`, which will contain `ExternalFunc1.c` and `ExternalFunc2.c` files. We should then create object files and place them in the archive, which will be an external library. To do this we use the following command's list:

```
gcc -c -o ExternalFunc1.o ExternalFunc1.c
gcc -c -o ExternalFunc2.o ExternalFunc2.c
ar rcs libExternalFunc1.a ExternalFunc1.o
ar rcs libExternalFunc2.a ExternalFunc2.o
```

To create object files, we use gcc with `-c` option and the archiver `ar` to place generated object files in the archive. As a result, we get two of the file `libExternalFunc1.a` and `libExternalFunc2.a`. There is also the possibility to put all the needed object files in a single archive.

To call external functions, we must use the keyword `external`. The name of the wrapper function in Modelica language can be differ from the name of the external function. In this case, we must explicitly specify which external functions should be wrapped.

```
model ExternalLibraries
    // Function name differs
    function ExternalFunc1
        input Real x;
        output Real y;
        // Explicitly specifying C-function name
        external y=ExternalFunc1_ext(x) annotation(Library="ExternalFunc1");
    end ExternalFunc1;
```

```

function ExternalFunc2
  input Real x;
  output Real y;
  // The functions names are the same
  external "C" annotation(Library="ExternalFunc2");
end ExternalFunc2;

Real x(start=1.0, fixed=true), y(start=2.0, fixed=true);
equation
  der(x)=-ExternalFunc1(x);
  der(y)=-ExternalFunc2(y);
end ExternalLibraries;

```

Note that in the annotation the name of the external library is specified as `ExternalFunc1`, while the file itself is called `libExternalFunc1.a`. This is not a mistake and the prefix `lib` must be added to all library's files.

The example shows that the type `Real` corresponds to the C type `double`. Additionally, the types of `Integer` and `Boolean` match the C-type `int`. Arrays of type `Real` and `Integer` transferred in arrays of type `double` and `int`.

It should be noted that consistently works only call c-functions with arguments of `int` and `double` types, as well as arrays of these types. The attempt to use specific c-type, for example, `long long int` or an unsigned type such as `unsigned int`, causes the error.

B. Modeling stochastic Wiener process

Let us describe the implementation of a generator of the normal distribution and Wiener process. We assume that the generator of uniformly-distributed random numbers is already implemented in the functions `urand`. To generate the normal distribution we will use Box-Muller transformation and Wiener process can be calculated as cumulative sums of normally-distributed numbers.

The minimum working version of the code is shown below. The key point is the use of an operator `sample(t_0, h)`, which generates events using `h` seconds starting from the time `t_0`. For every event the operator `sample` calls the function `urand` that returns a new random number.

```

model generator
  Integer x1, x2;
  Port rnd; "Random number generator's port"
  Port normal; "Normal numbers generator's port"
  Port wiener; "Wiener process values port"
  Integer m = 429496729; "Generator modulo"
  Real u1, u2;
initial equation
  x1 = 114561;
  x2 = 148166;
algorithm
  when sample(0, 0.1) then
    x1 := urand(x1);
    x2 := urand(x2);
  end when;
  // normalisation of random sequence
  rnd.data[1] := x1 / m;
  rnd.data[2] := x2 / m;
  u1 := rnd.data[1];
  u2 := rnd.data[2];
  // normal generator
  normal.data[1] := sqrt(-2 * log(u1)) * sin(6.28 * u2);
  normal.data[2] := sqrt(-2 * log(u1)) * cos(6.28 * u2);
  // Wiener process
  wiener.data[1] := wiener.data[1] + normal.data[1];
  wiener.data[2] := wiener.data[2] + normal.data[2];

```

```
end generator;
```

Note also the use of a special variable of type Port which serves to connect the various models together. In our example we have created three such variables: `lg`, `normal`, `wiener`. Because of this, other models can access the result of our generator.

```
connector Port
  Real data[2];
end Port;
```

A minimal working code below illustrates the connection example between two models. A system of two ordinary differential equations describes van der Pol–Duffing oscillator with additive stochastic part in the form of a Wiener process (see 5).

$$\begin{cases} \dot{x} = y, \\ \dot{y} = x(1.0 - x^2) - y + x \cdot W_t. \end{cases}$$

It is important to mention that this equation is not stochastic. Built-in OpenModelica numerical methods do not allow to solve stochastic differential equations.

```
// the model specifies a system of ODE
model ODE
  Real x, y;
  Port IN;
initial equation
  x = 2.0;
  y = 0.0;
equation
  der(x) = y ;
  der(y) = x*(1-x*x) - y + x*IN.data[1];
end ODE;
model sim
  generator gen;
  ODE eq;
equation
  connect(gen.wiener, eq.IN);
end sim;
```

V. CONCLUSION

We reviewed the basic algorithms for generating uniformly distributed pseudo-random numbers. All algorithms were implemented by the authors in C language and tested using DieHarder utility. The test results revealed that the most effective algorithms are `xorshift` and Mersenne Twister algorithms.

Due to the fact that OpenModelica does not implement bitwise logical and shifting operators, generators of uniformly distributed pseudo-random numbers have to be implemented in C language and connected to the program as external functions. We gave a rather detailed description of this process, that, as we hope, will fill a gap in the official documentation.

ACKNOWLEDGMENTS

The work is partially supported by RFBR grants No’s 15-07-08795 and 16-07-00556. Also the publication was supported by the Ministry of Education and Science of the Russian Federation (the Agreement No 02.A03.21.0008). The computations were carried out on the Felix computational cluster (RUDN University, Moscow, Russia) and on

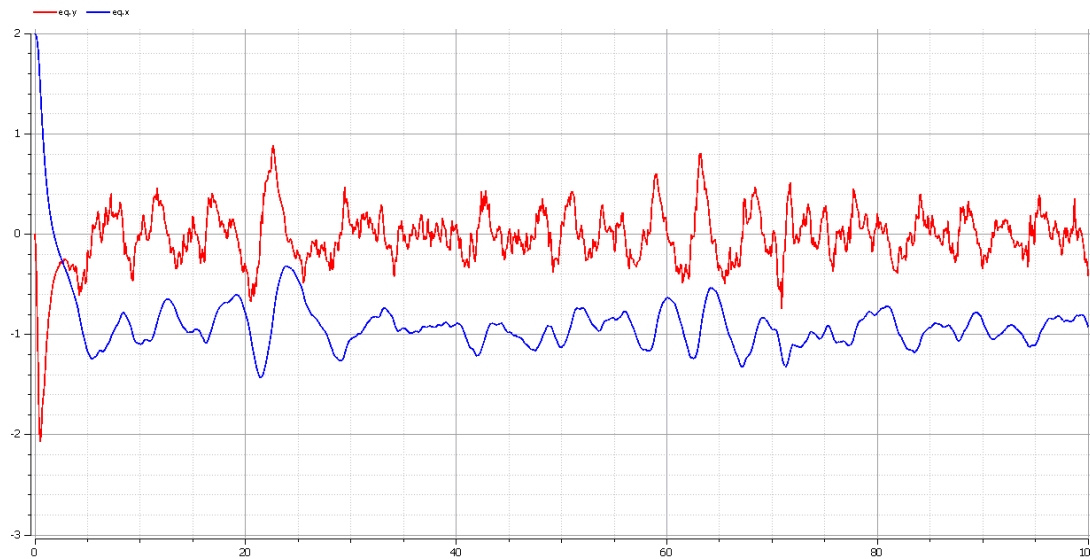


Figure 5. Results of van der Pol-Duffing oscillator simulation. The graphs are created using the functionality OMEditor's

the HybriLIT heterogeneous cluster (Multifunctional center for data storage, processing, and analysis at the Joint Institute for Nuclear Research, Dubna, Russia).

-
- [1] OpenModelica official site.
URL <https://www.openmodelica.org/>
 - [2] Modelica and the Modelica Association official site.
URL <https://www.modelica.org/>
 - [3] SciLab site.
URL <http://www.scilab.org/>
 - [4] LMS Imagine.Lab Amesim.
URL http://www.plm.automation.siemens.com/en_us/products/lms/imagine-lab/amesim/index.shtml
 - [5] Multi-Engineering Modeling and Simulation - Dymola - CATIA.
URL <http://www.3ds.com/products-services/catia/products/dymola>
 - [6] Jmodelica.org.
URL <http://www.jmodelica.org/>
 - [7] Wolfram SystemModeler.
URL <http://www.wolfram.com/system-modeler/index.ru.html>
 - [8] MapleSim - High Performance Physical Modeling and Simulation - Technical Computing Software.
URL <http://www.maplesoft.com/products/maplesim/index.aspx>
 - [9] A. Nishida, A. Fujii, Y. Oyanagi, Lis: Library of Iterative Solvers for Linear Systems.
URL http://www.phy.duke.edu/~rgb/General/rand_rate.php
 - [10] LAPACK, Linear Algebra PACKage.
URL <http://www.netlib.org/lapack/>
 - [11] SuiteSparse : a suite of sparse matrix software.
URL <http://faculty.cse.tamu.edu/davis/suitesparse.html>
 - [12] A. M. Collier, A. C. Hindmarsh, R. Serban, C. S. W. dward, User Documentation for KINSOL v2.8.2 (2015).
URL http://computation.llnl.gov/sites/default/files/public/kin_guide.pdf
 - [13] R. G. Brown, D. Eddelbuettel, D. Bauer, Dieharder: A Random Number Test Suite (2013).
URL http://www.phy.duke.edu/~rgb/General/rand_rate.php
 - [14] D. E. Knuth, The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms, Vol. 2, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
 - [15] P. L'Ecuyer, Tables of linear congruential generators of different sizes and good lattice structure, Mathematics of Computation 68 (225) (1999) 249–260.
 - [16] M. Galassi, B. Gough, G. Jungman, J. Theiler, J. Davies, M. Booth, F. Rossi, The GNU Scientific Library Reference Manual (2015).
URL <https://www.gnu.org/software/gsl/manual/gsl-ref.pdf>

- [17] P. L'Ecuyer, Combined multiple recursive random number generators, *Operations Research* 44 (5) (1996) 816–822.
- [18] P. L'Ecuyer, F. Blouin, R. Couture, A search for good multiple recursive random number generators, *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 3 (2) (1993) 87–98.
- [19] M. Matsumoto, T. Nishimura, Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Trans. Model. Comput. Simul.* 8 (1) (1998) 3–30. doi:10.1145/272991.272995.
URL <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf>
- [20] T. gfortran team, Python 3.5.1 documentation (Mar 2016).
URL <https://docs.python.org/3/>
- [21] G. Marsaglia, Xorshift rngs, *Journal of Statistical Software* 8 (1) (2003) 1–6. doi:10.18637/jss.v008.i14.
URL <https://www.jstatsoft.org/index.php/jss/article/view/v008i14>
- [22] F. Panneton, P. L'Ecuyer, On the xorshift random number generators, *ACM Trans. Model. Comput. Simul.* 15 (4) (2005) 346–361.
URL <http://doi.acm.org/10.1145/1113316.1113319>
- [23] G. Rose, Kiss: A bit too simple (2011).
URL <https://eprint.iacr.org/2011/007.pdf>
- [24] T. gfortran team, Using GNU Fortran (2015).
URL <https://gcc.gnu.org/onlinedocs/>
- [25] P. L'Ecuyer, R. Simard, Testu01: A c library for empirical testing of random number generators, *ACM Transactions on Mathematical Software (TOMS)* 33 (4) (2007) 22.
URL <http://www.iro.umontreal.ca/~lecuyer/myftp/papers/testu01.pdf>
- [26] E. Platen, N. Bruti-Liberati, *Numerical Solution of Stochastic Differential Equations with Jumps in Finance*, Springer, Heidelberg Dordrecht London New York, 2010.
- [27] G. E. P. Box, M. E. Muller, A note on the generation of random normal deviates, *The Annals of Mathematical Statistics* 29 (2) (1958) 610–611.
URL <http://dx.doi.org/10.1214/aoms/1177706645>
- [28] J. R. Bell, Algorithm 334: Normal random deviates, *Commun. ACM* 11 (7) (1968) 498–. doi:10.1145/363397.363547.
URL <http://doi.acm.org/10.1145/363397.363547>
- [29] R. Knop, Remark on algorithm 334 [g5]: Normal random deviates, *Commun. ACM* 12 (5) (1969) 281–. doi:10.1145/362946.362996.
URL <http://doi.acm.org/10.1145/362946.362996>
- [30] L. Devroye, *Non-Uniform Random Variate Generation*, Springer-Verlag, New York, 1986.
- [31] J. H. Ahrens, U. Dieter, Computer methods for sampling from gamma, beta, poisson and binomial distributions, *Computing* 12 (3) (1974) 223–246. doi:10.1007/BF02293108.
URL <http://dx.doi.org/10.1007/BF02293108>
- [32] J. H. Ahrens, U. Dieter, Computer generation of poisson deviates from modified normal distributions, *ACM Trans. Math. Softw.* 8 (2) (1982) 163–179. doi:10.1145/355993.355997.
URL <http://doi.acm.org/10.1145/355993.355997>
- [33] P. E. Kloeden, E. Platen, *Numerical Solution of Stochastic Differential Equations*, 2nd Edition, Springer, Berlin Heidelberg New York, 1995.
- [34] B. Øksendal, *Stochastic differential equations. An introduction with applications*, 6th Edition, Springer, Berlin Heidelberg New York, 2003.
- [35] A. Klöckner, F. L. J. van der Linden, D. Zimmer, Noise generation for continuous system simulation, in: *Proceedings of the 10th International Modelica Conference*, Lund; Sweden, 2014, pp. 837–846.
URL <http://www.ep.liu.se/ecp/096/087/ecp14096087.pdf>

Генерация стохастических процессов в OpenModelica

М. Н. Геворкян,^{1,*} А. В. Демидова,^{1,†} А. В. Королькова,^{1,‡} Д. С. Кулябов,^{1,2,§} и Л. А. Севастьянов^{1,3,¶}

¹*Кафедра прикладной информатики и теории вероятностей,
Российский университет дружбы народов,
ул. Миклухо-Маклая, д. 6, Москва, Россия, 117198*

²*Лаборатория информационных технологий,
Объединённый институт ядерных исследований,
ул. Жолио-Кюри 6, Дубна, Московская область, Россия, 141980*

³*Лаборатория теоретической физики,
Объединённый институт ядерных исследований,
ул. Жолио-Кюри 6, Дубна, Московская область, Россия, 141980*

Предпосылки: Язык компонентно-ориентированного моделирования **Modelica** применяется для моделирования сложных процессов, задаваемых системой ОДУ. В стандартной библиотеке **OpenModelica** нет средств для генерации псевдослучайных чисел, что делает невозможным моделирование стохастических процессов.

Цель: Целью данной статьи является краткий обзор некоторого числа алгоритмов генерации последовательности равномерно распределённых случайных чисел и оценка качества даваемых ими последовательностей, а также описание способов реализации некоторых из этих алгоритмов в системе **OpenModelica**.

Методы: Все рассматриваемые алгоритмы реализованы на языке C, а результаты их работы протестированы с помощью свободно распространяемых тестов **DieHarder**. Описывается реализация алгоритмов на языке **Modelica** и подключение виде C функций.

Результаты: Реализовано и протестировано около девяти алгоритмов. С помощью **DieHarder** выявлены наиболее качественные генераторы псевдослучайных чисел. Рассмотрены возможности библиотек для **OpenModelica**: **Noise** и **AdvancedNoise**.

Выводы: В системе **OpenModelica** могут быть реализованы генераторы равномерно распределённых псевдослучайных чисел, что является первым шагом на пути использования **OpenModelica** для моделирования стохастических процессов.

Ключевые слова: **Modelica**, **OpenModelica**, генератор случайных величин, винеровский процесс, пуассоновский процесс, СДУ

* gevorgyan_mn@rudn.university

† demidova_av@rudn.university

‡ korolkova_av@rudn.university

§ kulyabov_ds@rudn.university

¶ sevastianov_la@rudn.university

I. ВВЕДЕНИЕ

В данной статье рассматривается вопрос генерации равномерно распределенных псевдослучайных чисел, а также винеровского и пуассоновского стохастических процессов в среде OpenModelica [1]. OpenModelica является одной из открытых реализацией языка Modelica [2] (существуют также другие реализации данного языка [3–8]). Данный язык предназначен для моделирования различных систем и процессов, которые можно представить в виде системы алгебраических или дифференциальных уравнений. Для численного решения уравнений используется ряд открытых библиотек [9–12]. На данный момент, однако, в стандартной библиотеке OpenModelica нет средств даже для генерации равномерно распределенных случайных чисел.

В первой части статьи дается обзор некоторых алгоритмов генерации псевдослучайных чисел, в том числе описывается псевдоустройство `/dev/random` ОС Unix. Для большинства из них приводится алгоритм в виде псевдокода. Все алгоритмы реализованы авторами на языке C и частично на языке OpenModelica, а также протестированы с помощью пакета тестов `dieharder`. На основе результатов тестирования выбираются лучшие алгоритмы для использования.

Во второй части статьи приводятся алгоритмы генерации нормального и пуассоновского распределений на основе генераторов равномерно распределенных псевдослучайных чисел. Даются краткие сведения из теории случайных процессов, аксиоматические определения пуассоновского и винеровского процессов, а также алгоритмы, позволяющие генерировать эти процессы на компьютере.

Третья часть статьи носит практическую направленность и посвящена описанию вызова функций на языке C непосредственно из кода программы на языке Modelica

II. АЛГОРИТМЫ ГЕНЕРАЦИИ РАВНОМЕРНО РАСПРЕДЕЛЕННЫХ ПСЕВДОСЛУЧАЙНЫХ ЧИСЕЛ

В данном разделе мы опишем несколько наиболее распространенных генераторов равномерно распределенных псевдослучайных чисел. Такие генераторы служат основой для получения последовательностей псевдослучайных чисел других распределений.

A. Линейный конгруэнтный метод

Линейный конгруэнтный метод был впервые предложен в 1949 году Д. Г. Лехмером (D. H. Lehmer) [13]. Алгоритм II.1 задается одной формулой:

$$x_{n+1} = (ax_n + c) \bmod m, \quad n \geq 0,$$

где m — *модуль* (mask) $m > 1$, a — *множитель* (multiplier) ($0 \leq a < m$), c — *приращение* ($0 \leq c < m$), x_0 — начальное значение, *зерно* (seed). Результатом многократного применения данной рекуррентной формулы является *линейная конгруэнтная последовательность* x_1, \dots, x_n . Особый случай $c = 0$ называется *мультипликативным* конгруэнтным методом. Для краткого обозначения данного метода будем использовать аббревиатуру LCG (linear congruential generator).

Алгоритм II.1 LCG линейный конгруэнтный генератор

Require: $n, seed$
 const $m \leftarrow 2^{64}$
 const $a \leftarrow 6364136223846793005$
 const $c \leftarrow 1442695040888963407$
 $x_0 \leftarrow seed$
for $i = 0$ to n **do**
 $x_i = (a \cdot x_{i-1} + c) \bmod m$
end for
return $\{x_0, x_1, \dots, x_n\}$

Числа m , a , c называют «волшебными» или «магическими» так как их значения задаются в коде программы и выбираются исходя из опыта применения генератора. Качество генерируемой последовательности существенно зависит от правильного выбора данных параметров. Последовательность $\{x\}_1^n$ периодична и ее период зависит

от числа m , которое поэтому должно быть большим. На практике выбирают m равным машинному слову (для 32-х битной архитектуры — 2^{32} и для 64-х битной — 2^{64}). В [13] рекомендуется выбрать

$$a = 6364136223846793005, c = 1442695040888963407, m = 2^{64} = 18446744073709551616.$$

В статье [14] можно найти объемные таблицы с оптимальными значениями a , b и m .

Квадратичный конгруэнтный метод $x_n = (ax_{n-1}^2 + bx_{n-1} + d) \bmod m$ кубический конгруэнтный метод $x_n = (ax_{n-1}^3 + bx_{n-1}^2 + cx_{n-1} + d) \bmod 2^e$.

В настоящее время линейный конгруэнтный метод представляет по большей части лишь исторический интерес, так как он генерирует сравнительно некачественную псевдослучайную последовательность по сравнению с другими, не менее простыми генераторами.

Авторы реализовали линейный конгруэнтный метод II.1 на языке C и сгенерировали с помощью него последовательность из 10^9 чисел. Данная последовательность была протестирована с помощью открытого набора тестов DieHarder [15]. В результате генератор LCG провалил около половины тестов.

В. Метод Фибоначчи с запаздываниями

Развитием LCG генератора можно считать идею использовать для генерации i -го элемента псевдослучайной последовательности не один, а несколько предыдущих элементов. Согласно [13] первый такой генератор был предложен в начале 50-х годов и основывался на формуле:

$$x_{n+1} = (x_n + x_{n-1}) \bmod m.$$

Однако на практике он показал себя не лучшим образом. В 1958 году Дж. Ж. Митчелом (G. J. Mitchell) и Д. Ф. Муром (D. Ph. Moore) был изобретен намного лучший генератор II.2

$$x_n = (x_{n-n_a} + x_{n-n_b}) \bmod m, n \geq \max(n_a, n_b).$$

Данный генератор получил название генератора Фибоначчи с запаздыванием (LFG, lagged Fibonacci Generator).

Алгоритм II.2 LFG генератор Фибоначчи с запаздываниями

```

 $n_a \leftarrow 55$ 
 $n_b \leftarrow 24$ 
Require:  $s_0, s_1, \dots, s_{n_b}, n \geq 0$ 
 $x_0, x_1, \dots, x_{n_b} \leftarrow r_0, r_1, \dots, r_{n_b}$ 
for  $i = (n_a + 1)$  to  $n$  do
  if  $x_{i-n_a} \geq x_{i-n_b}$  then
     $x_i = x_{i-n_a} - x_{i-n_b}$ 
  else if  $x_{i-n_a} < x_{i-n_b}$  then
     $x_i = x_{i-n_a} - x_{i-n_b} + 1$ 
  end if
end for
return  $\{x_0, x_1, \dots, x_n\}$ 

```

Как и в случае LCG генератора, выбор «магических чисел» n_a и n_b сильно влияет на качество генерируемой последовательности. Авторы предложили использовать следующие магические числа n_a и n_b :

$$n_a = 24, n_b = 55.$$

Д. Кнут [13, 16] приводит ряд других значений, начиная от (37, 100) и заканчивая (9739, 23209) Длина периода данного генератора в точности равна $2^{e-1}(2^{55} - 1)$ при выборе $m = 2^e$.

Как видно из алгоритма, для инициализации данного генератора необходимо использовать не одно начальное значение, а последовательность из $\max(n_a, n_b)$ случайных чисел.

В открытой библиотеке GNU Scientific Library (GSL) [17] используется *составной мульти-рекурсивный* генератор, предложенный в статье [18]. Данный генератор является разновидностью LFG и может быть задан следующими формулами:

$$\begin{aligned}
 x_n &= (a_1 x_{n-1} + a_2 x_{n-2} + a_3 x_{n-3}) \bmod m_1, \\
 y_n &= (b_1 y_{n-1} + b_2 y_{n-2} + b_3 y_{n-3}) \bmod m_2, \\
 z_n &= (x_n - y_n) \bmod m_1.
 \end{aligned}$$

Составной характер данного алгоритма позволяет получить большой период, равный $10^{56} \approx 2^{185}$. В GSL используются следующие значения параметров a_i, b_i, m_1, m_2 :

$$\begin{aligned} a_1 &= 0, & b_1 &= 86098, & m_1 &= 2^{32} - 1 = 2147483647, \\ a_2 &= 63308, & b_2 &= 0, & m_2 &= 2145483479, \\ a_3 &= -183326, & b_3 &= -539608. \end{aligned}$$

Еще один метод, предложенный в статье [19] также является разновидностью метода Фибоначчи и определяется формулой:

$$x_n = (a_1 x_{n-1} + a_5 x_{n-5}) \mod 5,$$

В GSL использованы следующие параметры: $a_1 = 107374182$, $a_2 = 0$, $a_3 = 0$, $a_4 = 0$, $a_5 = 104480$, $m = 2^{31} - 1 = 2147483647$. Период этого генератора равен 10^{46} .

Генератор LFG II.2 был реализован авторами на языке C и подвергнут тестированию с помощью DieHarder [15]. Генератор показал высокое качество сгенерированной последовательности (10^9 чисел), что дает основание использовать его при моделировании стохастических процессов.

С. Инверсный конгруэнтный генератор

Инверсный конгруэнтный метод основан на использовании обратного по модулю числа.

$$x_{i+1} = (ax_i^{-1} + b) \mod m$$

где a — множитель ($0 \leq a < n$), b — приращение ($0 \leq b < n$), x_0 — начальное значение (seed). Кроме того $\text{НОД}(x_0, m) = 1$ и $\text{НОД}(a, m) = 1$.

Данный генератор превосходит обычный линейный метод, однако сложнее алгоритмически, так как необходимо искать обратные по модулю целые числа, что приводит к медленной скорости генерации чисел. Для вычисления обратного числа обычно применяется расширенный алгоритм Евклида [13, §4.3.2].

Д. Генераторы с использованием побитовых операций

Большинство генераторов, дающих наиболее качественные псевдослучайные последовательности используют в своих алгоритмах побитовые операции конъюнкции, дизъюнкции, отрицания, исключающей дизъюнкции (xor) и побитовые вправо/влево.

1. Вихрь Мерсенна

Считается одним из лучших псевдослучайных генераторов. Разработан в 1997 году Мацумото и Нишимура [20]. Существуют 32-, 64-, 128-разрядные версии вихря Мерсенна. Свое название алгоритм получил из-за использования простого числа Мерсенна $2^{19937} - 1$. В зависимости от реализации обеспечивается период вплоть до $2^{216091} - 1$.

Основным недостатком алгоритма является относительная громоздкость и, как следствие, сравнительно медленная работа. В остальном же данный генератор обеспечивает качественную псевдослучайную последовательность и проходит все тесты DieHarder. Важным преимуществом является требование лишь одного инициализирующего числа (seed). Вихрь Мерсенна используется во многих стандартных библиотеках, например в модуле random языка Python 3 [21].

Ввиду громоздкости алгоритма мы не приводим его псевдокод в данной статье, однако стандартная имплементация алгоритма, созданная Мацумото и Нишимура свободно доступна по ссылке <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt64.html>.

2. Генераторы xorShift

Несколько простых генераторов (алгоритмы II.3 и II.4, дающих качественную псевдослучайную последовательность) были разработаны в 2003 году Дж. Марсальей (G. Marsaglia) [22, 23].

Алгоритм II.3 Генератор xorshift***Require:** $n, seed$ $x \leftarrow seed$ $y_0 \leftarrow x$ **for** $i = 1$ to n **do** $x \leftarrow x \oplus x \gg 12$ $x \leftarrow x \oplus x \ll 25$ $x \leftarrow x \oplus x \gg 27$ $y_i \leftarrow x \cdot 2685821657736338717$ **end for****return** $\{y_0, y_1, \dots, y_n\}$ **Алгоритм II.4** Генератор xorshift+**Require:** $n, seed_1, seed_2$ **for** $i = 1$ to n **do** $x \leftarrow seed_1$ $y \leftarrow seed_2$ $seed_1 \leftarrow y$ $x = x \oplus (x \ll 23)$ $seed_2 = x \oplus y \oplus (x \gg 17) \oplus (y \gg 26)$ $z_i \leftarrow seed_2 + y$ **end for****return** $\{z_1, \dots, z_n\}$

3. Генераторы KISS (Keep It Simple Stupid)

Еще одно семейство генераторов, дающих не менее качественную последовательность псевдослучайных чисел [24]. Генератор KISS используется в процедуре `random_number()` языка `Frotran` (компилятор `gfortran` [25])

Алгоритм II.5 Генератор KISS**Require:** $n, seed_0, seed_1, seed_2, seed_3$ t **for** $i = 1$ to n **do** $seed_0 \leftarrow 69069 \cdot seed_0 + 123456$ $seed_1 \leftarrow seed_1 \oplus (seed_1 \ll 13)$ $seed_1 \leftarrow seed_1 \oplus (seed_1 \gg 17)$ $seed_1 \leftarrow seed_1 \oplus (seed_1 \ll 5)$ $t \leftarrow 698769069 \cdot seed_2 + seed_3$ $seed_3 \leftarrow (t \gg 32)$ $seed_1 \leftarrow t$ $x_i \leftarrow seed_0 + seed_1 + seed_2$ **end for****return** $\{x_1, \dots, x_n\}$ **Алгоритм II.6** Генератор jKISS**Require:** $n, seed_0, seed_1, seed_2, seed_3$ t **for** $i = 1$ to n **do** $seed_0 \leftarrow 314527869 \cdot seed_0 + 1234567$ $seed_1 \leftarrow seed_1 \oplus (seed_1 \ll 5)$ $seed_1 \leftarrow seed_1 \oplus (seed_1 \gg 7)$ $seed_1 \leftarrow seed_1 \oplus (seed_1 \ll 22)$ $t \leftarrow 4294584393 \cdot seed_2 + seed_3$ $seed_3 \leftarrow (t \gg 32)$ $seed_1 \leftarrow t$ $x_i \leftarrow seed_0 + seed_1 + seed_2$ **end for****return** $\{x_1, \dots, x_n\}$ **Е. Устройства /dev/random и /dev/urandom**

Перевод комментариев к драйверу `random.c`. <https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/drivers/char/random.c?id=refs/tags/v3.15.6#n52>.

Для создания истинно-случайной последовательности чисел с помощью компьютера, в некоторых Unix системах (в частности GNU/Linux) используется сбор «фоновых шумов» окружения операционной системы и аппаратного обеспечения. Источником такого случайного шума являются моменты времени между нажатия клавиш пользователем (inter-keyboard timings), различные системные прерывания и другие события, которые удовлетворяют двум требованиям: не детерминированности и сложной доступности для измерения внешним наблюдателем.

Неопределённость из таких источников собирается драйвером ядра и помещается в «энтропийный пул», который дополнительно перемешивается с помощью алгоритма, похожего на алгоритмы вычисления контрольных сум. Когда случайные байты запрашиваются системным вызовом, они извлекаются из энтропийного пула путем взятия SHA хеша от содержимого пула. Взятие хеша позволяет не показывать внутреннее состояние пула, так как восстановление содержимого по хешу считается вычислительно невыполнимой задачей. Дополнительно извлекающая процедура занижает размер содержимого пула для того, чтобы предотвратить выдачу хеша по всему содержимому и минимизировать теоретическую возможность определения его содержимого.

Во вне энтропийный пул доступен в виде символьного псевдоустройства `/dev/random`, а также в виде системного вызова:

```
void get_random_bytes(void *buf, int nbytes);
% \end{minted}
```

Устройство `/dev/random` можно использовать для получения очень качественной последовательности случайных чисел, однако оно возвращает число байт, равное размеру накопленного энтропийного пула, поэтому если

требуется неограниченное количество случайных чисел, то следует использовать символьное псевдоустройство `/dev/urandom` у которого нет данного ограничения, но оно уже генерирует псевдослучайные числа, качество которых достаточно для большинства не криптографических задач.

Г. Тестирование алгоритмов

Обзор большого числа критериев оценки качества массива псевдослучайных чисел можно найти в третьей главе книги Д. Кнута [13], а также статье [26] одного из ведущих специалистов по генераторам псевдослучайных чисел. Все описанные в нашей статье алгоритмы были реализованы на языке C и протестированы с помощью набора тестов `dieharder`, доступного на официальном сайте автора [15]. Также этот пакет тестов входит в состав официальных репозиториях многих дистрибутивов GNU/Linux.

1. Описание *dieharder*

Набор тестов `dieharder` реализован в виде утилиты командной строки, которая позволяет тестировать последовательности равномерно распределенных псевдослучайных чисел. Также `dieharder` может использовать любой генератор из библиотеки GSL [17] для генерирования чисел или непосредственного тестирования.

- `dieharder -l` — показать список доступных тестов,
- `dieharder -g -1` — показать список доступных генераторов псевдослучайных чисел, каждому генератору присвоено порядковое число, которое надо указать после флага `-g` для включения нужного генератора.
 - 200 `stdin_input_raw` — считывать стандартный входной бинарный поток,
 - 201 `file_input_raw` — считывать файл в бинарном формате,
 - 202 `file_input` — считывать файл в текстовом формате,
 - 500 `/dev/random` — использовать псевдо-устройство `/dev/random`,
 - 501 `/dev/urandom` — использовать псевдо-устройство `/dev/urandom`.

Каждое псевдослучайное число должно располагаться на новой строке, а в первых строках файла необходимо указать: тип чисел (`d` — целые числа двойной точности), количество чисел в файле и разрядность чисел (32 или 64 бита). Пример такого файла:

```
type: d
count: 5
numbit: 64
1343742658553450546
16329942027498366702
3111285719358198731
2966160837142136004
17179712607770735227
```

Когда такой файл создан можно передать его в `dieharder`

```
dieharder -a -g 202 -f file.in > file.out
```

где флаг `-a` включает все встроенные тесты, а флаг `-f` задает файл для анализа. Результаты тестирования будут сохранены в `file.out`.

2. Результаты тестов и выводы

Генератор	Провалено	Слабо	Пройдено
LCG (линейный конгруэнтный генератор)	52	6	55
LCG2 (составной линейный конгруэнтный генератор)	51	8	54
LFG (линейный генератор Фибоначчи)	0	2	111
ICG (инверсный конгруэнтный генератор)	0	6	107
KISS	0	3	110
jKISS	0	4	109
XorShift	0	4	109
XorShift+	0	2	111
XorShift*	0	2	111
MT (вихрь Мерсенна)	0	2	111
dev/urandom	0	2	111

Из генераторов, использующих побитовые операции выделяются генераторы `xorshift*`, `xorshift+` и вихрь Мерсенна. Все они дают одинаково качественную последовательность. Алгоритм вихря Мерсенна, однако, намного более громоздок, чем `xorshift*` или `xorshift+`, поэтому для генерирования больших последовательностей предпочтительней использовать `xorshift*` или `xorshift+`.

Среди генераторов не использующих побитовые операции выделяется линейный генератор Фибоначчи, который при тестировании дает результаты на уровне `XorShift+` и вихря Мерсенна. Однако необходимость задать минимум 55 начальных значений для инициализации данного генератора сводит его полезность к минимуму. Инверсный конгруэнтный генератор показывает немногим худшие результаты, но требует всего одно число для инициализации алгоритма.

III. АЛГОРИТМЫ ГЕНЕРАЦИИ ВИНЕРОВСКОГО И ПУАССОНОВСКОГО ПРОЦЕССОВ

Рассмотрим вопрос генерации нормального и пуассоновского распределений. Выбор именно этих двух распределений мотивирован их ключевой ролью в теории стохастических дифференциальных уравнений. Наиболее общий вид таких уравнений использует два случайных процесса: винеровский и пуассоновский [27]. Винеровский процесс позволяет учесть имплицитную стохастичность моделируемой системы, а пуассоновский процесс — внешнее воздействие. Кратко опишем структуру статьи.

A. Генерирование равномерно распределенных псевдослучайных чисел из единичного промежутка

Генераторы псевдослучайных равномерно распределенных чисел являются основой для получения других псевдослучайных последовательностей. Однако, большинство алгоритмов требуют задания случайного числа из интервала $[0, 1]$, в то время как подавляющее большинство генераторов равномерно распределенных псевдослучайных чисел дают последовательность из интервала $[0, m]$, где число m зависит от алгоритма, разрядности операционной системы и процессора. Чаще всего используют $m \approx 2^{32}$ для 32х битных и $m \approx 2^{64}$ для 64 битных систем.

Для получения чисел из интервала $[0, 1]$ можно поступить двумя способами. Во первых можно нормировать полученную последовательность случайных чисел, поделив каждый ее элемент на максимальный элемент последовательности. Такой подход гарантированно даст 1 в качестве одного из случайных чисел. Однако такой способ плох, когда последовательность псевдослучайных чисел слишком велика и не умещается в оперативную память. В этом случае лучше использовать второй способ, а именно поделить каждое сгенерированное число на m .

B. Генерирование нормального распределения

Метод генерации нормально распределенных псевдослучайных величин предложен в 1958 году Дж. Э. Р. Боксом и П. Мюллером [28] и назван в их честь *преобразованием Бокса-Мюллера*. Метод основан на простом порообразовании, задаваемом двумя формулами. Данное преобразование обычно записывается в двух форматах:

- стандартная форма (она как раз и предложена авторами статьи [28]),
- полярная форма (предложена Дж. Беллом [29] и Р. Кнопом [30]).

Стандартная форма. Пусть x и y — два независимых, равномерно распределенных псевдослучайных числа из интервала $(0, 1)$, тогда числа z_1 и z_2 вычисляются по формуле

$$z_1 = \cos(2\pi y)\sqrt{-2\ln x}, \quad z_2 = \sin(2\pi y)\sqrt{-2\ln x}$$

и являются независимыми псевдослучайными числами, распределенными по стандартному нормальному закону $\mathcal{N}(0, 1)$ с математическим ожиданием $\mu = 0$ и стандартным среднеквадратичным отклонением $\sigma = 1$.

Полярная форма. Пусть x и y — два независимых, равномерно распределенных псевдослучайных числа из интервала $[-1, 1]$. Вычислим вспомогательную величину $s = x^2 + y^2$, если $s > 1$ и $s = 0$, то значения x и y следует отбросить и проверить следующую пару. Если же $0 < s \leq 1$ тогда числа z_1 и z_2 вычисляются по формуле

$$z_1 = x\sqrt{\frac{-2\ln s}{s}}, \quad z_2 = y\sqrt{\frac{-2\ln s}{s}}$$

и являются независимыми псевдослучайными числами, распределенными по стандартному нормальному закону $\mathcal{N}(0, 1)$.

При компьютерной реализации данного алгоритма предпочтительней использовать полярную форму, так как в этом случае приходится вычислять только одну трансцендентную функцию \ln , а не три (\ln , \sin \cos), как в стандартном варианте. Это компенсирует даже тот факт, что часть исходных равномерно распределенных чисел отбрасывается — полярная версия метода все равно работает быстрее. Пример работы алгоритма изображен на рисунке 1

Для получения нормального распределения общего вида из стандартного нормального распределения используют формулу $Z = \sigma \cdot z + \mu$, где $z \sim \mathcal{N}(0, 1)$, а $Z \sim \mathcal{N}(\mu, \sigma)$.

С. Генерирование распределения Пуассона

Для генерирования распределения Пуассона существует большое число различных алгоритмов [31–33]. Наиболее простой был предложен Кнутом [13]. Для работы алгоритма III.1 необходимо уметь генерировать равномерные псевдослучайные числа из промежутка $[0, 1]$. Пример работы алгоритма изображен на рисунке 2

Алгоритм III.1 Генератор распределения Пуассона

Require: $seed, \lambda$

$\Lambda \leftarrow \exp(-\lambda), k \leftarrow 0, p \leftarrow 1, u \leftarrow seed$

repeat

$k \leftarrow k + 1$

$u \leftarrow rand(u)$

$p = p \cdot u$

until $p > \Lambda$

return $k - 1$

▷ генерируем равномерно распределенное случайное число

D. Генерирование пуассоновского и винеровского процессов

Вышеописанные генераторы нормального и пуассоновского распределений мы используем для генерации винеровского и пуассоновского стохастических процессов. Дадим вначале определения данных процессов, а затем перейдем к описанию алгоритмов.

1. Определение винеровского и пуассоновского случайных процессов

Пусть (Ω, \mathcal{A}, P) — вероятностное пространство, где Ω — пространство элементарных событий, \mathcal{A} — σ -алгебра подмножеств Ω (случайные события), P — вероятность или, иначе, вероятностная мера такая что $P(\Omega) = 1$.

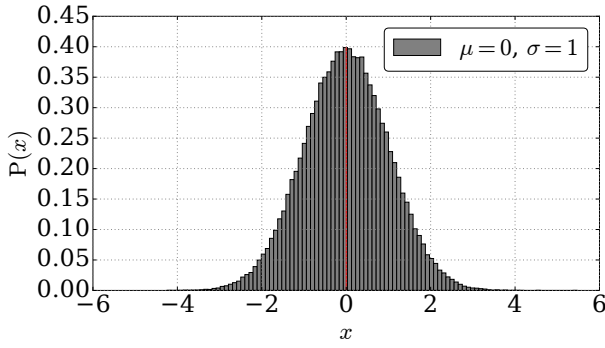


Рис. 1. Нормальное распределение

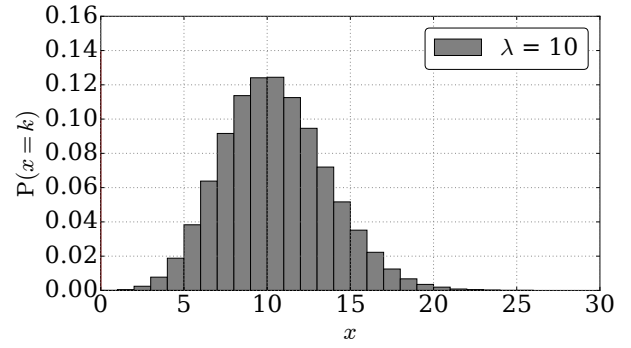


Рис. 2. Распределение Пуассона

Определение (см. [34, 35]). Семейство случайных величин $X = \{X_t, 0 \leq t \leq T\}$, где $X_t \in \mathbb{R}^d$ будет называться d -мерным стохастическим процессом, где совокупность их конечномерных функций распределения

$$F_{X_{t_1}, X_{t_2}, \dots, X_{t_k}}(x_{i_1}, x_{i_2}, \dots, x_{i_k}) = P(X_{t_1} \leq x_{i_1}, X_{t_2} \leq x_{i_2}, \dots, X_{t_k} \leq x_{i_k})$$

для всех $i_k = 1, 2, 3, \dots, k = 1, 2, 3, \dots, x_i \in \mathbb{R}^d$ и $t_k \in T$

Пространство состояний X — d -мерное евклидово пространство \mathbb{R}^d , $d = 1, 2, 3, \dots$. Временной интервал $[0, T]$, где $T > 0$. В численных методах рассматривают последовательность моментов времени $\{t_0, t_1, t_2, \dots\}$.

Определение (см. [34, 35]). Случайный кусочно-постоянный процесс $N = \{N_t, 0 \leq t \leq T\}$ с интенсивностью $\lambda > 0$ называется *процессом Пуассона* (пуассоновским) если выполняются следующие свойства.

1. $P\{N_0 = 0\} = 1$, иначе говоря $N_0 = 0$ почти наверное.
2. N_t — процесс с независимыми приращениями, то есть $\{\Delta N_0, \Delta N_1, \dots\}$ независимые случайные величины; $\Delta N_{t_i} = N_{t_{i+1}} - N_{t_i}$ и $0 \leq t_0 < t_1 < t_2 < \dots < t_n \leq T$; $\Delta N_{t_i} = N_{t_{i+1}} - N_{t_i}$ и $0 \leq t_0 < t_1 < t_2 < \dots < t_n \leq T$.
3. Существует число $\lambda > 0$ такое, что для любого приращения ΔN_i , $i = 0, \dots, n-1$, $E[\Delta N_i] = \lambda \Delta t_i$.
4. Если $P(s) = P\{N_{t+s} - N_t > 2\}$, то $\lim_{s \rightarrow 0} \frac{P(s)}{s} = 0$.

Определение (см. [34, 35]). Случайный процесс $W = \{W_t, 0 \leq t \leq T\}$ называется скалярным *процессом Винера* (винеровским) если выполняются следующие условия.

1. $P\{W_0 = 0\} = 1$, иначе говоря $W_0 = 0$ почти наверное.
2. W_t — процесс с независимыми приращениями, то есть $\{\Delta W_0, \Delta W_1, \dots\}$ независимые случайные величины; $\Delta W_{t_i} = W_{t_{i+1}} - W_{t_i}$ и $0 \leq t_0 < t_1 < t_2 < \dots < t_n \leq T$.
3. $\Delta W_i = W_{t_{i+1}} - W_{t_i} \sim \mathcal{N}(0, t_{i+1} - t_i)$ где $0 \leq t_{i+1} < t_i < T$, $i = 0, 1, \dots, n-1$.

Из определения следует, что ΔW_i — нормально распределённая случайная величина с математическим ожиданием $\mathbb{E}[\Delta W_i] = \mu = 0$ и дисперсией $\mathbb{D}[\Delta W_i] = \sigma^2 = \Delta t_i$.

Винеровский процесс является моделью *броуновского движения* (хаотического блуждания). Если рассмотреть процесс W_t в те моменты времени $0 = t_0 < t_1 < t_2 < \dots < t_{N-1} < t_N$ когда он испытывает случайные аддитивные изменения, то непосредственно из определения следует: $W_{t_1} = W_{t_0} + \Delta W_0$, $W_{t_2} = W_{t_1} + \Delta W_1$, ..., $W(t_N) = W(t_{N-1}) + \Delta W_{n-1}$, где $\Delta W_i \sim \mathcal{N}(0, \Delta t_i)$, $\forall i = 0, \dots, n-1$.

Запишем W_{t_n} в виде накопленной суммы: $W_{t_n} = W_{t_0} + \sum_{i=0}^{n-1} \Delta W_i$ и учтем, что $\mathbb{E}[\Delta W_i] = 0$ и $\mathbb{D}[\Delta W_i] = \Delta t_i$ можно показать, что сумма нормально распределенных случайных чисел ΔW_i также является нормально распределенным случайным числом:

$$\mathbb{E} \sum_{i=0}^{n-1} \Delta W_i = 0, \quad \mathbb{D} \sum_{i=0}^{n-1} \Delta W_i = \sum_{i=0}^{n-1} \Delta t_i = T, \quad \sum_{i=0}^{n-1} \Delta W_i \sim \mathcal{N}(0, T).$$

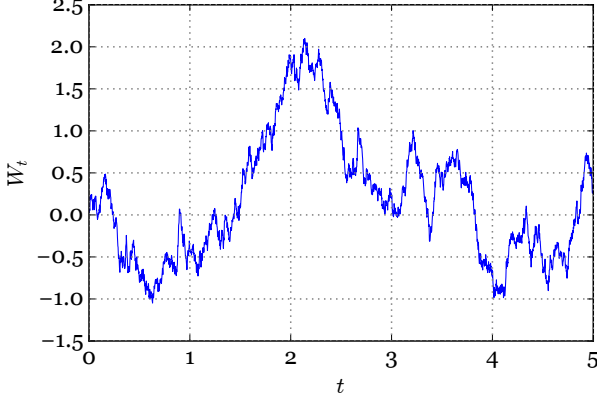


Рис. 3. Винеровский процесс

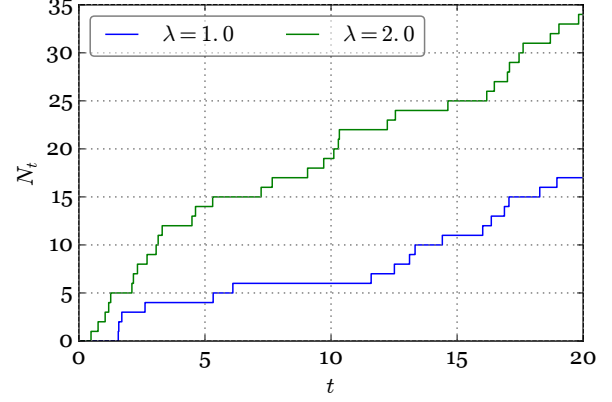


Рис. 4. Пуассоновский процесс

Многомерный винеровский процесс $\mathbf{W}_t: \Omega \times [t_0, T] \rightarrow \mathbb{R}^m$ определяют как случайный процесс составленный из совместно независимых одномерных винеровских процессов W_t^1, \dots, W_t^m . Приращения ΔW_t^α , $\forall \alpha = 1, \dots, m$ являются совместно независимыми нормально распределенными случайными величинами. С другой стороны вектор ΔW_t^α можно представить как многомерную нормально распределенную случайную величину с вектором математических ожиданий $\mu = 1$ и диагональной матрицей ковариаций.

2. Генерирование винеровского процесса

Для симулирования одномерного винеровского процесса необходимо сгенерировать N нормально распределенных случайных чисел $\varepsilon_1, \dots, \varepsilon_N$ и построить их накопленные суммы $\varepsilon_1, \varepsilon_1 + \varepsilon_2, \varepsilon_1 + \varepsilon_2 + \varepsilon_3 \dots$ В результате мы получим *выборочную траекторию* винеровского процесса $W(t)$ см. рис. 3.

В случае многомерного случайного процесса следует сгенерировать уже m последовательностей из N нормально распределенных случайных величин.

3. Генерирование пуассоновского процесса

Симулирование пуассоновского процесса во многом аналогично винеровскому, но теперь необходимо сгенерировать последовательность из чисел распределенных по пуассоновскому закону и затем вычислить их накопленную сумму. График процесса изображен на рис. 4. Из графика видно, что пуассоновский процесс представляет собой скачкообразное изменение числа произошедших с течением времени событий. От интенсивности λ зависит среднее количество событий за отрезок времени.

Из за такого характерного поведения пуассоновский процесс также называется скачкообразным, а стохастические дифференциальные уравнения где он участвует в качестве второго ведущего процесса получили названия уравнений со скачками [27]

IV. МОДЕЛИРОВАНИЕ СТОХАСТИЧЕСКИХ ПРОЦЕССОВ В OPENMODELICA

Как уже отмечалось во введении, в OpenModelica отсутствуют псевдослучайные генераторы, что делает эту систему непригодной для моделирования стохастических процессов. Стоит, однако, отметить библиотеку `Noise` build.openmodelica.org/Documentation/Noise.html, разрабатываемую Клёкнером (Klöckner) [36] и другими. В основе данной библиотеки лежит семейство генераторов `xorshift` (алгоритмы II.3 и II.4 из первого раздела), написанных на языке C. Однако неподготовленный пользователь может столкнуться с проблемой при использовании библиотеки `Noise`, так как для ее работы необходимо сперва скомпилировать исходные файлы библиотек на языке C.

В данной части статьи мы опишем процедуру подключения внешних C функций к OpenModelica, что позволит пользователю использовать как библиотеку `Noise`, так и подключить свои генераторы случайных чисел. Также мы

приведем минимальный рабочий пример генерации стохастического винеровского процесса в среде OpenModelica и решения обыкновенного дифференциального уравнения с аддитивно добавленным стохастическим членом.

А. Подключение внешних C функций к OpenModelica

Перейдем к рассмотрению процесса подключения внешних функций к программе на OpenModelica. Соответствующий раздел в официальной документации при описании данной процедуры упускает несколько существенных деталей и следование его указаниям приводит к ошибке. Все описанные нами действия проводились на компьютере под управлением операционной системы Ubuntu Linux версии 16.04 LTS и OpenModelica версии 1.11.0-dev-15.

При компиляции кода OpenModelica программа транслируется в C-код, который затем обрабатывается уже C-компилятором. Поэтому поддержка C-функций реализована в OpenModelica на уровне архитектуры. Кроме языка C OpenModelica поддерживает также вызов Fortran (только F77) и Python функций. Однако оба этих языка поддерживаются косвенным путем, а именно с помощью обертывания их в соответствующие C-функции.

Использование внешних C-функций может понадобиться по разным причинам: повышение быстродействия ресурсоёмких компонент программы, необходимость использования полноценного императивного языка программирования, использование уже существующего C-кода т.д.

Приведем простейший пример вызова C функций из программы написанной на языке Modelica. Создадим два исходных файла `ExternalFunc1.c` и `ExternalFunc2.c`. В этих файлах будут содержаться простейшие функции, которые мы хотим использовать в нашей программе на языке Modelica.

```
// Файл ExternalFunc1.c
double ExternalFunc1_ext(double x)
{
    return x+2.0*x*x;
}

// Файл ExternalFunc2.c
double ExternalFunc2(double x)
{
    return (x-1.0)*(x+2.0);;
}
```

В директории, где находится исходный код программы на языке Modelica необходимо создать каталог `Resources` а в нем каталог `Library`, в котором будут находиться оба наших файла `ExternalFunc1.c` и `ExternalFunc2.c`. После этого следует создать объектные файлы и поместить их в архив, который будет служить подключаемой библиотекой. Для этого следует выполнить ряд команд.

```
gcc -c -o ExternalFunc1.o ExternalFunc1.c
gcc -c -o ExternalFunc2.o ExternalFunc2.c
ar rcs libExternalFunc1.a ExternalFunc1.o
ar rcs libExternalFunc2.a ExternalFunc2.o
```

Для создания объектных файлов мы использовали утилиту компиляции `gcc` с опцией `-c` и архиватор `ar` для помещения созданных объектных файлов в архив. В результате мы получим два файла `libExternalFunc1.a` и `libExternalFunc2.a`. Есть также возможность поместить все необходимые нам объектные файлы в один архив.

Для вызова внешних функций следует использовать ключевое слово `external`. Название функции-обертки на языке Modelica может как совпадать, так и отличаться от названия внешней функции. Во втором случае необходимо явно указать какую внешнюю функции следует обернуть.

```
model ExternalLibraries
    // Название функции не совпадает с названием функции
    // на языке C
    function ExternalFunc1
        input Real x;
        output Real y;
        // Явно указываем название C-функции
        external y=ExternalFunc1_ext(x) annotation(Library="ExternalFunc1");
    end ExternalFunc1;
```

```

function ExternalFunc2
  input Real x;
  output Real y;
  // Названия функций совпадают, поэтому явное указание
  // не требуется
  external "C" annotation(Library="ExternalFunc2");
end ExternalFunc2;

Real x(start=1.0, fixed=true), y(start=2.0, fixed=true);
equation
  der(x)=-ExternalFunc1(x);
  der(y)=-ExternalFunc2(y);
end ExternalLibraries;

```

Заметьте, что в аннотации название подключаемой библиотеки указано как `ExternalFunc1`, в то время как сам файл называется `libExternalFunc1.a`. Это не является опечаткой и приставку `lib` необходимо добавлять ко всем библиотечным файлам.

Из примера видно, что тип `Real` языка Modelica соответствует типу `double` языка C. Кроме того тип `Integer` и `Boolean` соответствуют типу `int`. Массивы типа `Real` и `Integer` также переводятся в массивы типа `double` и `int`.

Стабильно работает вызов функций с аргументами типа `int` и `double`, а также массивами этих типов. Попытка же уточнить используемый тип, например `long long int` или использовать беззнаковый тип, например `unsigned int` приводит к ошибке.

В. Моделирование стохастического винеровского процесса

Опишем реализацию генератора нормального распределения и винеровского процесса. Будем предполагать, что генератор равномерно-распределенных случайных величин уже реализован в виде функции `urand`. Для генерирования нормального воспользуемся вышеописанным преобразованием Бокса-Мюллера, а элементы последовательности, задающей винеровский процесс вычислим как кумулятивную сумму нормально-распределенных чисел.

Минимальный рабочий вариант кода приведен ниже. Ключевым моментом является использование оператора `sample(t_0, h)`, который генерирует события через `h` секунд, начиная с момента времени `t_0`. При каждом срабатывании оператора `sample` вызывается функция `urand`, которая возвращает новое случайное число.

```

model generator
  Integer x1, x2;
  Port lg; "Порт для линейного генератора"
  Port normal; "Порт для нормально распределенных чисел"
  Port wiener; "Порт для винеровского процесса"
  Integer m = 429496729; "Значение модуля (маски), применяемого в генераторе"
  Real u1, u2;
initial equation
  x1 = 114561;
  x2 = 148166;
algorithm
  when sample(0, 0.1) then
    x1 := urand(x1);
    x2 := urand(x2);
  end when;
  // если необходимо, то нормируем псевдослучайные числа
  lg.data[1] := x1 / m;
  lg.data[2] := x2 / m;
  u1 := lcg.data[1];
  u2 := lcg.data[2];
  // нормальный генератор
  normal.data[1] := sqrt(-2 * log(u1)) * sin(6.28 * u2);

```

```

normal.data[2] := sqrt(-2 * log(u1)) * cos(6.28 * u2);
// Винеровский процесс
wiener.data[1] := wiener.data[1] + normal.data[1];
wiener.data[2] := wiener.data[2] + normal.data[2];
end generator;

```

Отметим также применение специальной переменной типа Port, которая служит для соединения различных моделей между собой. В нашем примере мы создали три таких переменных: `lg`, `normal`, `wiener`. Благодаря этому, другие модели могут получить доступ к результату работы нашего генератора.

```

connector Port
  Real data[2];
end Port;
% \end{minted}

```

Ниже представлен минимальный код примера иллюстрирующего соединение моделей между собой. В качестве примера выбрана система из двух обыкновенных дифференциальных уравнений, описывающая осциллятор Ван-дер-Поля–Дуффинга и к ней добавлен стохастический вклад в виде процесса Винера (см. 5).

$$\begin{cases} \dot{x} = y, \\ \dot{y} = x(1.0 - x^2) - y + x \cdot W_t. \end{cases}$$

Отметим особо, что это уравнение не является стохастическим. Встроенные в OpenModelica численные методы не позволяют решать стохастические дифференциальные уравнения.

```

// модель задает систему ОДУ
model ODE
  Real x, y;
  Port IN;
  initial equation
    x = 2.0;
    y = 0.0;
  equation
    der(x) = y ;
    der(y) = x*(1-x*x) - y + x*IN.data[1];
  end ODE;
// Эту модель надо запускать на симуляцию
// в ней числа из случайного генератора поступают
// в систему ОДУ
model sim
  generator gen;
  ODE eq;
  equation
    connect(gen.wiener, eq.IN);
  end sim;

```

V. ЗАКЛЮЧЕНИЕ

Был дан обзор основным алгоритмам генерации равномерно распределенных псевдослучайных чисел. Все алгоритмы были реализованы авторами на языке C и протестированы с помощью утилиты DieHarder. Результаты тестов выявили, что наиболее эффективными алгоритмами являются алгоритмы `xorshift` и вихрь Мерсенна.

В связи с тем, что в OpenModelica не реализованы побитовые логические операции и операции сдвига, генераторы равномерно распределенных псевдослучайных чисел следует реализовать на языке C и подключить в виде внешних функций. Мы дали достаточно подробное описание процесса подключения внешних C-функций, восполняющее пробелы официальной документации.

Был приведён минимальный рабочий пример моделирования винеровского стохастического процесса, который был использован для моделирования осциллятор Ван-дер-Поля–Дуффинга с аддитивным стохастическим членом. Важно отметить, что такое уравнение не является стохастическим. Для построения моделей на основе стохастических уравнений среду OpenModelica необходимо дополнить стохастическими численными методами, для чего по видимому необходимо внести изменения в исходный код OpenModelica.

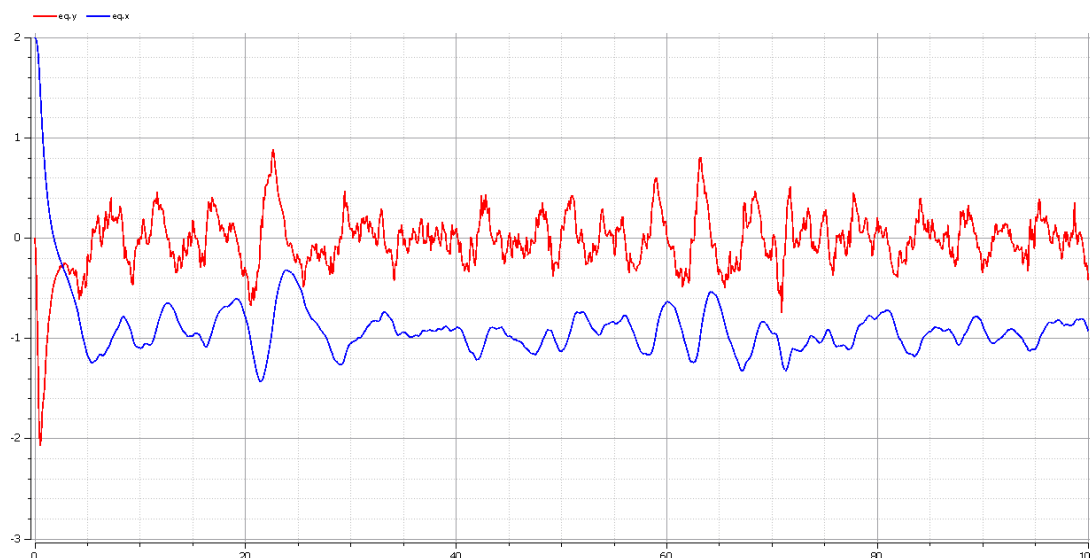


Рис. 5. Результаты моделирования осциллятора Ван-дер-Поля–Дуффинга с добавленным стохастическим вкладом. Графики созданы с помощью функционала OMEditor'a

БЛАГОДАРНОСТИ

Работа частично поддержана грантами РФФИ № 15-07-08795, 16-07-00556. Также публикация выполнена при финансовой поддержке Минобрнауки России (Соглашение № 02.А03.21.0008). Расчёты проведены на вычислительном кластере «Felix» РУДН и на Гетерогенном вычислительном кластере «HybriLIT» Многофункционального центра хранения, обработки и анализа данных ОИЯИ.

-
- [1] OpenModelica official site. — URL: <https://www.openmodelica.org/> (online; accessed: 19.06.2016).
 - [2] Modelica and the Modelica Association official site. — URL: <https://www.modelica.org/> (online; accessed: 19.06.2016).
 - [3] SciLab site. — URL: <http://www.scilab.org/> (online; accessed: 19.06.2016).
 - [4] LMS Imagine.Lab Amesim. — URL: http://www.plm.automation.siemens.com/en_us/products/lms/Imagine-Lab/amesim/index.shtml (online; accessed: 19.06.2016).
 - [5] Multi-Engineering Modeling and Simulation - Dymola - CATIA. — URL: <http://www.3ds.com/products-services/catia/products/dymola> (online; accessed: 19.06.2016).
 - [6] Jmodelica.org. — URL: <http://www.jmodelica.org/> (online; accessed: 19.06.2016).
 - [7] Wolfram SystemModeler. — URL: <http://www.wolfram.com/system-modeler/index.ru.html> (online; accessed: 19.06.2016).
 - [8] MapleSim - High Performance Physical Modeling and Simulation - Technical Computing Software. — URL: <http://www.maplesoft.com/products/maplesim/index.aspx> (online; accessed: 19.06.2016).
 - [9] Nishida A., Fujii A., Oyanagi Y. Lis: Library of Iterative Solvers for Linear Systems. — URL: http://www.phy.duke.edu/~rgb/General/rand_rate.php (online; accessed: 19.06.2016).
 - [10] LAPACK, Linear Algebra PACKage. — URL: <http://www.netlib.org/lapack/> (online; accessed: 19.06.2016).
 - [11] SuiteSparse : a suite of sparse matrix software. — URL: <http://faculty.cse.tamu.edu/davis/suitesparse.html> (online; accessed: 19.06.2016).
 - [12] Collier A. M., Hindmarsh A. C., Serban R., dward C. S. W. User Documentation for KINSOL v2.8.2. — 2015. — URL: http://computation.llnl.gov/sites/default/files/public/kin_guide.pdf.
 - [13] Кнут Д. Э. Искусство программирования. — 3 изд. — Москва : Вильямс, 2004. — Т. 2. — ISBN: 5-8459-0081-6.
 - [14] L'Ecuyer P. Tables of linear congruential generators of different sizes and good lattice structure // Mathematics of Computation. — 1999. — Vol. 68, no. 225. — P. 249–260.
 - [15] Brown R. G., Eddelbuettel D., Bauer D. Dieharder: A Random Number Test Suite. — 2013. — URL: http://www.phy.duke.edu/~rgb/General/rand_rate.php.
 - [16] Knuth D. E. The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1997. — Vol. 2. — ISBN: 0-201-89684-2.
 - [17] Galassi M., Gough B., Jungman G. et al. — The GNU Scientific Library Reference Manual, 2015. — URL: <https://www.gnu.org/software/gsl/>

- [//www.gnu.org/software/gsl/manual/gsl-ref.pdf](http://www.gnu.org/software/gsl/manual/gsl-ref.pdf).
- [18] L'Ecuyer P. Combined multiple recursive random number generators // *Operations Research*. — 1996. — Vol. 44, no. 5. — P. 816–822.
 - [19] L'Ecuyer P., Blouin F., Couture R. A search for good multiple recursive random number generators // *ACM Transactions on Modeling and Computer Simulation (TOMACS)*. — 1993. — Vol. 3, no. 2. — P. 87–98.
 - [20] Matsumoto M., Nishimura T. Mersenne twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator // *ACM Trans. Model. Comput. Simul.* — 1998. — January. — Vol. 8, no. 1. — P. 3–30. — URL: <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf>.
 - [21] gfortran team T. — Python 3.5.1 documentation, 2016. — Mar. — URL: <https://docs.python.org/3/>.
 - [22] Marsaglia G. Xorshift rngs // *Journal of Statistical Software*. — 2003. — Vol. 8, no. 1. — P. 1–6. — URL: <https://www.jstatsoft.org/index.php/jss/article/view/v008i14>.
 - [23] Panneton F., L'Ecuyer P. On the xorshift random number generators // *ACM Trans. Model. Comput. Simul.* — 2005. — October. — Vol. 15, no. 4. — P. 346–361. — URL: <http://doi.acm.org/10.1145/1113316.1113319>.
 - [24] Rose G. KISS: A Bit Too Simple. — 2011. — URL: <https://eprint.iacr.org/2011/007.pdf>.
 - [25] gfortran team T. — Using GNU Fortran, 2015. — URL: <https://gcc.gnu.org/onlinedocs/>.
 - [26] L'Ecuyer P., Simard R. Testu01: A C library for empirical testing of random number generators // *ACM Transactions on Mathematical Software (TOMS)*. — 2007. — Vol. 33, no. 4. — P. 22. — URL: <http://www.iro.umontreal.ca/~lecuyer/myftp/papers/testu01.pdf>.
 - [27] Platen E., Bruti-Liberati N. *Numerical Solution of Stochastic Differential Equations with Jumps in Finance*. — Heidelberg Dordrecht London New York : Springer, 2010.
 - [28] Box G. E. P., Muller M. E. A note on the generation of random normal deviates // *The Annals of Mathematical Statistics*. — 1958. — 06. — Vol. 29, no. 2. — P. 610–611. — URL: <http://dx.doi.org/10.1214/aoms/1177706645>.
 - [29] Bell J. R. Algorithm 334: Normal Random Deviates // *Commun. ACM*. — 1968. — July. — Vol. 11, no. 7. — P. 498–. — URL: <http://doi.acm.org/10.1145/363397.363547>.
 - [30] Knop R. Remark on algorithm 334 [g5]: Normal Random Deviates // *Commun. ACM*. — 1969. — May. — Vol. 12, no. 5. — P. 281–. — URL: <http://doi.acm.org/10.1145/362946.362996>.
 - [31] Devroye L. *Non-Uniform Random Variate Generation*. — New York : Springer-Verlag, 1986.
 - [32] Ahrens J. H., Dieter U. Computer methods for sampling from gamma, beta, poisson and binomial distributions // *Computing*. — 1974. — Vol. 12, no. 3. — P. 223–246. — URL: <http://dx.doi.org/10.1007/BF02293108>.
 - [33] Ahrens J. H., Dieter U. Computer generation of poisson deviates from modified normal distributions // *ACM Trans. Math. Softw.* — 1982. — June. — Vol. 8, no. 2. — P. 163–179. — URL: <http://doi.acm.org/10.1145/355993.355997>.
 - [34] Kloeden P. E., Platen E. *Numerical Solution of Stochastic Differential Equations*. — 2 edition. — Berlin Heidelberg New York : Springer, 1995. — ISBN: 3-540-54062-8.
 - [35] Øksendal B. *Stochastic differential equations. An introduction with applications*. — 6 edition. — Berlin Heidelberg New York : Springer, 2003. — ISBN: 3-540-04758-1.
 - [36] Klöckner A., van der Linden F. L. J., Zimmer D. Noise generation for continuous system simulation // *Proceedings of the 10th International Modelica Conference*. — Lund; Sweden, 2014. — P. 837–846. — URL: <http://www.ep.liu.se/ecp/096/087/ecp14096087.pdf>.