Vladimir M. Vishnevskiy
Konstantin E. Samouylov
Dmitry V. Kozyrev (Eds.)

Communications in Computer and Information Science 678

Distributed Computer and Communication Networks

19th International Conference, DCCN 2016
Moscow, Russia, November 21–25, 2016
Revised Selected Papers

Springer

Vladimir M. Vishnevskiy · Konstantin E. Samouylov
Dmitry V. Kozyrev (Eds.)

Distributed Computer and Communication Networks

19th International Conference, DCCN 2016
Moscow, Russia, November 21–25, 2016
Revised Selected Papers

Springer

Editors

Vladimir M. Vishnevskiy
V.A. Trapeznikov Institute of Control Sciences
Russian Academy of Sciences
Moscow
Russia

Konstantin E. Samouylov
RUDN University
Moscow
Russia

Dmitry V. Kozyrev
V.A. Trapeznikov Institute of Control Sciences
Russian Academy of Sciences
Moscow
Russia

and

RUDN University
Moscow
Russia

ISSN 1865-0929 ISSN 1865-0937 (electronic)
Communications in Computer and Information Science
ISBN 978-3-319-51916-6 ISBN 978-3-319-51917-3 (eBook)
DOI 10.1007/978-3-319-51917-3

Library of Congress Control Number: 2016963656

Preface

This volume contains a collection of revised selected full-text papers presented at the 19th International Conference on Distributed Computer and Communication Networks (DCCN 2016), held in Moscow, Russia, November 21–25, 2016.

The conference is a continuation of traditional international conferences of the DCCN series, which took place in Bulgaria (Sofia, 1995, 2005, 2006, 2008, 2009, 2014), Israel (Tel Aviv, 1996, 1997, 1999, 2001), and Russia (Moscow, 1998, 2000, 2003, 2007, 2010, 2011, 2013, 2015) in the past 19 years. The main idea of the conference is to provide a platform and forum for researchers and developers from academia and industry from various countries working in the area of theory and applications of distributed computer and communication networks, mathematical modeling, methods of control and optimization of distributed systems, by offering them a unique opportunity to share their views, discuss prospective developments, and pursue collaborations in this area. The content of this volume is related to the following subjects:

1. Computer and communication networks architecture optimization
2. Control in computer and communication networks
3. Performance and QoS/QoE evaluation in wireless networks
4. Analytical modeling and simulation of next-generation communications systems
5. Queuing theory and reliability theory applications in computer networks
6. Wireless 4G/5G networks, cm- and mm-wave radio technologies
7. RFID technology and its application in intellectual transportation networks
8. Internet of Things, wearables, and applications of distributed information systems
9. Probabilistic and statistical models in information systems
10. Mathematical modeling of high-tech systems
11. Mathematical modeling and control problems
12. Distributed and cloud computing systems, big data analytics

The DCCN 2016 conference gathered 208 submissions from authors from 20 different countries. From these, 141 high-quality papers in English were accepted and presented during the conference, 56 of which were recommended by session chairs and selected by the Program Committee for the Springer proceedings.

All the papers selected for the proceedings are given in the form presented by the authors. These papers are of interest to everyone working in the field of computer and communication networks.

We thank all the authors for their interest in DCCN, the members of the Program Committee for their contributions, and the reviewers for their peer-reviewing efforts.

November 2016

Vladimir M. Vishnevskiy
Konstantin E. Samouylov

Organization

DCCN 2016 was jointly organized by the Russian Academy of Sciences (RAS), the V. A. Trapeznikov Institute of Control Sciences of RAS (ICS RAS), the Peoples' Friendship University of Russia (RUDN), the National Research Tomsk State University, and the Institute of Information and Communication Technologies of Bulgarian Academy of Sciences (IICT BAS).

Steering Committee

General Chairs

S.N. Vasilyev	ICS RAS, Russia
V.M. Filippov	RUDN University, Russia
V.M. Vishnevskiy	ICS RAS, Russia
K.E. Samouylov	RUDN University, Russia

Program Committee

G. Adam	Joint Institute for Nuclear Research, Romania
A.M. Andronov	Transport and Telecommunication Institute, Latvia
E.A. Ayrjan	Joint Institute for Nuclear Research, Armenia
L.I. Abrosimov	Moscow Power Engineering Institute, Russia
Mo Adda	University of Portsmouth, UK
T.I. Aliev	ITMO University, Russia
S.D. Andreev	Tampere University of Technology, Finland
G. Araniti	University Mediterranea of Reggio Calabria, Italy
Bijan Saha	Joint Institute for Nuclear Research, Bangladesh
J.Busa	Technical University of Košice (TUKE), Slovakia
H. Chaouchi	Institut Télécom SudParis, France
T. Czachorski	Institute of Informatics of the Polish Academy of Sciences, Poland
B.N.Chetverushkin	Keldysh Institute of Applied Mathematics of RAS, Russia
O. Chuluunbaatar	National University of Mongolia, Mongolia
A.N. Dudin	Belarusian State University, Belarus
D. Fiems	Ghent University, Belgium
V.P. Gerdt	Joint Institute for Nuclear Research, Russia
A. Gelman	IEEE Communications Society, USA
D. Grace	York University, UK
A.A. Grusho	Federal Research Center "Computer Science and Control" of RAS, Russia
M. Hnatich	Pavol Jozef Šafárik University in Košice (UPJŠ), Slovakia
J. Hošek	Brno University of Technology, Czech Republic
J. Kolodziej	Cracow University of Technology, Poland

V.Y. Korolev	Lomonosov Moscow State University, Russia
B. Khoromskij	Max Planck Institute for Mathematics in the Sciences, Germany
C. Kim	Sangji University, Korea
G. Kotsis	Johannes Kepler University Linz, Austria
A. Krishnamoorthy	Cochin University of Science and Technology, India
A.E. Kucheryavy	Bonch-Bruevich St. Petersburg State University of Telecommunications, Russia
E.A. Kucheryavy	Tampere University of Technology, Finland
L. Lakatos	Budapest University, Hungary
R. Lazarov	Texas A&M University, USA
E. Levner	Holon Institute of Technology, Israel
B.Y. Lemeshko	Novosibirsk State Technical University, Russia
S.D. Margenov	Institute of Information and Communication Technologies of Bulgarian Academy of Sciences, Bulgaria
O. Martikainen	Service Innovation Research Institute, Finland
L. Militano	University Mediterranea of Reggio Calabria, Italy
E.V. Morozov	Institute of Applied Mathematical Research of the Karelian Research Centre RAS, Russia
G.K. Mishkoy	Academy of Sciences of Moldova, Moldavia
A.A. Nazarov	Tomsk State University, Russia
I. Novak	Brno University of Technology, Czech Republic
D.A. Novikov	ICS RAS, Russia
Y.N. Orlov	Keldysh Institute of Applied Mathematics of RAS, Russia
M. Pagano	Pisa University, Italy
I.V. Puzynin	Joint Institute for Nuclear Research, Russia
Y.P. Rybakov	RUDN University, Russia
V.V. Rykov	Gubkin Russian State University of Oil and Gas, Russia
Z. Saffer	Budapest University of Technology and Economics, Hungary
L.A. Sevastianov	RUDN University, Russia
S.Ya. Shorgin	Federal Research Center "Computer Science and Control" of RAS, Russia
A.L. Skubachevskii	RUDN University, Russia
P. Stanchev	Kettering University, USA
A.M. Turlikov	St. Petersburg State University of Aerospace Instrumentation, Russia
D. Udumyan	University of Miami, USA
S.I. Vinitsky	Joint Institute for Nuclear Research, Russia
J.P. Zaychenko	Kyiv Polytechnic Institute, Ukraine

Executive Committee

D.V. Kozyrev (Chair)	RUDN University and ICS RAS, Russia
S.P. Moiseeva	Tomsk State University, Russia
T. Atanasova	IICT BAS, Bulgaria

Organizers and Partners

Organizers

Russian Academy of Sciences
RUDN University
V.A. Trapeznikov Institute of Control Sciences of RAS (ICS RAS)
National Research Tomsk State University (NR TSU)
Institute of Information and Communication Technologies of Bulgarian Academy of Sciences (IICT-BAS)
Research and Development Company "Information and Networking Technologies"

Support

Information support was provided by the Moscow department of the IEEE Communication Society. Financial support was provided by the Russian Foundation for Basic Research.

Contents

The Stochastic Processes Generation in OpenModelica

Migran Gevorkyan[1], Michal Hnatich[3,4,5], Ivan M. Gostev[6], A.V. Demidova[1], Anna V. Korolkova[1], Dmitry S. Kulyabov[1,2], and Leonid A. Sevastianov[1,3(✉)]

[1] Department of Applied Probability and Informatics, RUDN University (Peoples' Friendship University of Russia), 6 Miklukho-Maklaya str., Moscow 117198, Russia
{mngevorkyan,avdemidova,akorolkova,dharma,sevast}@sci.pfu.edu.ru
[2] Laboratory of Information Technologies, Joint Institute for Nuclear Research, 6 Joliot-Curie, Dubna, Moscow Region 141980, Russia
hnatic@saske.sk
[3] Bogoliubov Laboratory of Theoretical Physics, Joint Institute for Nuclear Research, 6 Joliot-Curie, Dubna, Moscow Region 141980, Russia
[4] Department of Theoretical Physics, SAS, Institute of Experimental Physics, Watsonova 47, 040 01 Košice, Slovakia
[5] Faculty of Science, Pavol Jozef Šafárik University in Košice (UPJŠ), Šrobárova 2, 041 80 Košice, Slovakia
[6] National Research University Higher School of Economics, 20 Myasnitskaya Ulitsa, Moscow 101000, Russia
igostev@hse.ru

Abstract. This paper studies program implementation problem of pseudo-random number generators in OpenModelica. We give an overview of generators of pseudo-random uniform distributed numbers. They are used as a basis for construction of generators of normal and Poisson distributions. The last step is the creation of Wiener and Poisson stochastic processes generators. We also describe the algorithm to call external C-functions from programs written in Modelica. This allows us to use random number generators implemented in the C language.

Keywords: Modelica · OpenModelica · Random generator · Wiener process · Poisson process · SDE

1 Introduction

In this article we study the problem of generation of uniformly distributed pseudo-random numbers, stochastic Wiener and Poisson processes in OpenModelica framework [7]. OpenModelica is one of the open source implementations of Modelica [5] modeling language (for other implementations see [1,3,4,6,8,10]). 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 [2,9,18,31]. 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 pseudo-random numbers generation, 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 C language and partly in OpenModelica. Also we tested them with dieharder—a random number generator testing suite [17].

In the second part of the paper we describe algorithms for normal and Poisson distributions generation. 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 external functions (written in C language) calling directly from OpenModelica programs code.

2 Algorithms for Uniformly Distributed Pseudo-random Numbers Generating

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.

2.1 Linear Congruential Generator

A linear congruential generator (LCG) was first proposed in 1949 by Lehmer [24]. The algorithm is given by the formula:

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

where m is *the mask* or *the modulus* $m > 1$, a is *the multiplier* $(0 \leqslant a < m)$, c is *the increment* $(0 \leqslant c < m)$, 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.

The numbers m, a, c are called “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}). Knuth [24] recommends to choose

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

In the article [26], you can find large tables with optimal values a, b 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.

2.2 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 [24] 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, it showed itself not the best way. In 1958 George. J. Mitchell and D. Ph. Moore invented a much better generator

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

It was the generator that we now call `LFG`—**l**agged **F**ibonacci **G**enerator.

As in the case of `LCG` generator the "magical numbers" n_a and n_b greatly affect 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 [24] 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 an initial value and a sequence of $\max(n_a, n_b)$ random numbers must be used for the initialization of this generator.

In open source GNU Scientific Library (GSL) [20] the *composite multi-recursive* generator is used. It was proposed in paper [25]. This generator is a generalisation of `LFG` and 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 of a_i, b_i, m_1, m_2:

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

Another method suggested in the paper [27] 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}.

2.3 Inverse Congruential Generator

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

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

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

This generator is superior to the usual linear method, however, it is more complicated algorithmically, since it is necessary to find the inverse modulo integers which leads to performance reduction. The extended Euclidean algorithm [24, §4.3.2] is usually applied for compution of the inverse of the number.

2.4 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.

Mersenne Twister. Mersenne twister is considered to be one of the best pseudo-random generators. It was developed in 1997 by Matsumoto and Nishimura [30]. 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` [12].

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.

XorShift Generator. Some simple generators giving a high quality pseudo-random sequence were developed in 2003 by George. Marsala (Marsaglia) [29,33].

KISS Generator. Another group of generators, giving a high quality sequence of pseudo-random numbers, is KISS generators family [35] (Keep It Simple Stupid). They are used in the procedure `random_number()` of `Frotran` language (`gfortran` compiler [11]).

2.5 Pseudo Devices `/dev/random` and `/dev/urandom`

To create a truly random sequence of numbers using a computer some Unix systems (in particular GNU/Linux) use 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 is 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 its 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);
```

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.

2.6 Algorithms Testing

A review of quality criterias for a sequence of pseudo-random numbers can be found in the third chapter of the book [24], as well as in paper [28]. 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 [17].

Dieharder Overview. Dieharder is a 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 [20] 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 pseudo-random 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.

Table 1. Test results

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

Test Results and Conclusions. The best generators with bitwise operations are `xorshift*`, `xorshift+` and Mersenne Twister (see Table 1). 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 was showed by 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 its usefulness is reduced to a minimum. Inverse congruential generator shows slightly worse results, but requires only one number to initiate the algorithm.

3 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 [34]. Wiener process allows to take into account the implicit stochasticity of the simulated system, and the Poisson process—external influence.

3.1 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 its 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.

3.2 Normal Distribution Generation

An algorithm for normal distributed numbers generation has been proposed in 1958 by Bux and Mueller [16] 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 [16]),
– polar form (suggested by Bell [15] and Knop [23]).

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

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

and they 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 $0 < s \geqslant 1$ then the numbers z_1 and z_2 are calculated according to the formula

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

and they 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 Fig. 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 $Z \sim \mathcal{N}(\mu, \sigma)$.

3.3 The Generation of a Poisson Distribution

To generate a Poisson distribution there is a wide variety of algorithms [13, 14, 19]. The easiest was proposed by Knut [24]. This Algorithm 2.1 uses uniform pseudo-random number from the interval $[0, 1]$ for it's work. The algorithm's output example is depicted on Fig. 2.

Algorithm 2.1. The generator of the Poisson distribution

```
Require: $seed$, $\lambda$
  $\Lambda \leftarrow \exp(-\lambda)$, $k \leftarrow 0$, $p \leftarrow 1$, $u \leftarrow seed$
  repeat
      $k \leftarrow k + 1$
      $u \leftarrow rand(u)$              ▷ generation of uniformly distributed random number
      $p = p \cdot u$
  until $p > \Lambda$
  return $k - 1$
```

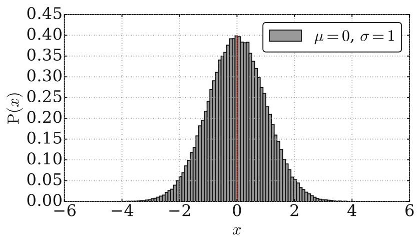

Fig. 1. Normal distribution

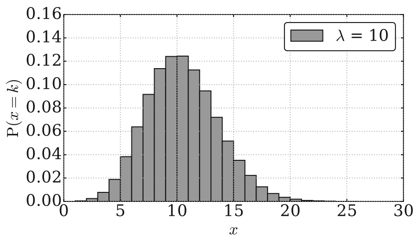

Fig. 2. Poisson distribution

3.4 Generation of Poisson and Wiener Processes

Now we going to use generators of normal and Poisson distributions to generate Wiener and Poisson stochastic processes. For definitions of Poisson and Wiener processes see, for example, [22,32,34].

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 $W(t)$ see Fig. 3.

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

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 that has occurred over time events. The intensity λ depends on the average number of events over a period of time.

Because of this characteristic of behavior the Poisson process is also called as a process with jumps, and stochastic differential equations, with Poisson process as second driving process, are called equations with jumps [34]

4 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 Klockner (Klockner) [21] should be mentioned. The basis of this library are `xorshift` generators, written in C. However, an inexperienced user may face a problem, because one needs compile C-source files first to use that library.

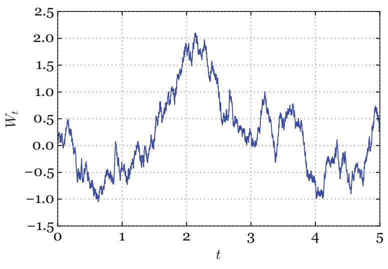

Fig. 3. Wiener process

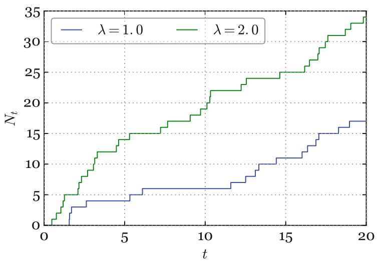

Fig. 4. Poisson process

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.

4.1 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 the 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 usage of external C-functions may be required for various reasons, for example, implementations of performance requiring components of the program, the usage of 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 `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 ExternalFunc1 // Function name differs
    input Real x;
    output Real y;
    external y=ExternalFunc1_ext(x); // Explicitly specifying C-function name
    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 -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.

4.2 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 PolDuffing 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 Open-Modelica 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;
```

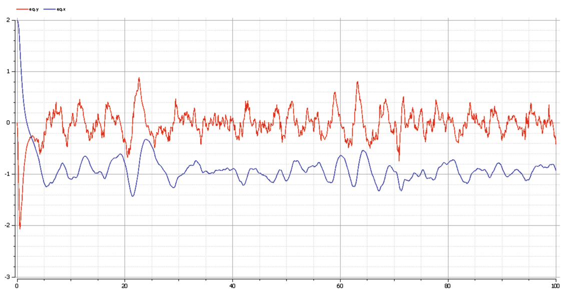

Fig. 5. Results of van der Pol–Duffing oscillator simulation. The graphs are created using the functionality OMEditor'a

5 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 14-01-00628, 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).

References

1. Jmodelica.org. http://www.jmodelica.org/
2. LAPACKLinear Algebra PACKage. http://www.netlib.org/lapack/
3. LMS Imagine.Lab Amesim. http://www.plm.automation.siemens.com/en_us/products/lms/imagine-lab/amesim/index.shtml
4. MapleSim - High Performance Physical Modeling and Simulation - Technical Computing Software. http://www.maplesoft.com/products/maplesim/index.aspx
5. Modelica and the Modelica Association Official Site. https://www.modelica.org/
6. Multi-Engineering Modeling and Simulation - Dymola - CATIA. http://www.3ds.com/products-services/catia/products/dymola
7. OpenModelica Official Site. https://www.openmodelica.org/
8. SciLab Official Site. http://www.scilab.org/
9. SuiteSparse: A Suite of Sparse Matrix Software. http://faculty.cse.tamu.edu/davis/suitesparse.html
10. Wolfram SystemModeler. http://www.wolfram.com/system-modeler/index.html
11. Using GNU Fortran (2015). https://gcc.gnu.org/onlinedocs/
12. Python 3.5.1 Documentation, March 2016. https://docs.python.org/3/
13. Ahrens, J.H., Dieter, U.: Computer methods for sampling from gamma, beta, poisson and bionomial distributions. Computing **12**(3), 223–246 (1974)
14. Ahrens, J.H., Dieter, U.: Computer generation of poisson deviates from modified normal distributions. ACM Trans. Math. Softw. **8**(2), 163–179 (1982)
15. Bell, J.R.: Algorithm 334: normal random deviates. Commun. ACM **11**(7), 498 (1968)
16. Box, G.E.P., Muller, M.E.: A note on the generation of random normal deviates. Ann. Math. Stat. **29**(2), 610–611 (1958)
17. Brown, R.G., Eddelbuettel, D., Bauer, D.: Dieharder: A Random Number Test Suite (2013). http://www.phy.duke.edu/~rgb/General/rand_rate.php
18. Collier, A.M., Hindmarsh, A.C., Serban, R., Dward, C.S.W.: User Documentation for KINSOL v2.8.2 (2015). http://computation.llnl.gov/sites/default/files/public/kin_guide.pdf

19. Devroye, L.: Non-Uniform Random Variate Generation. Springer-Verlag, New York (1986)
20. Galassi, M., Gough, B., Jungman, G., Theiler, J., Davies, J., Booth, M., Rossi, F.: The GNU Scientific Library Reference Manual (2015). https://www.gnu.org/software/gsl/manual/gsl-ref.pdf
21. Klckner, A., van der Linden, F.L.J., Zimmer, D.: Noise generation for continuous system simulation. In: Proceedings of the 10th International Modelica Conference, Lund, Sweden, pp. 837–846 (2014)
22. Kloeden, P.E., Platen, E.: Numerical Solution of Stochastic Differential Equations, 2nd edn. Springer, Heidelberg (1995)
23. Knop, R.: Remark on algorithm 334 [g5]: normal random deviates. Commun. ACM **12**(5), 281 (1969)
24. Knuth, D.E.: The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms, vol. 2. Addison-Wesley Longman Publishing Co. Inc., Boston (1997)
25. L'Ecuyer, P.: Combined multiple recursive random number generators. Oper. Res. **44**(5), 816–822 (1996)
26. L'Ecuyer, P.: Tables of linear congruential generators of different sizes and good lattice structure. Math. Comput. **68**(225), 249–260 (1999)
27. L'Ecuyer, P., Blouin, F., Couture, R.: A search for good multiple recursive random number generators. ACM Trans. Modeling Comput. Simul. (TOMACS) **3**(2), 87–98 (1993)
28. L'Ecuyer, P., Simard, R.: Testu01: AC library for empirical testing of random number generators. ACM Trans. Mathe. Softw. (TOMS) **33**(4), 22 (2007)
29. Marsaglia, G.: Xorshift RNGs. J. Stat. Softw. **8**(1), 1–6 (2003)
30. Matsumoto, M., Nishimura, T.: Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1), 3–30 (1998)
31. Nishida, A., Fujii, A., Oyanagi, Y.: Lis: Library of Iterative Solvers for Linear Systems. http://www.phy.duke.edu/~rgb/General/rand_rate.php
32. Øksendal, B.: Stochastic Differential Equations: An Introduction with Applications, 6th edn. Springer, Heidelberg (2003)
33. Panneton, F., L'Ecuyer, P.: On the xorshift random number generators. ACM Trans. Model. Comput. Simul. **15**(4), 346–361 (2005)
34. Platen, E., Bruti-Liberati, N.: Numerical Solution of Stochastic Differential Equations with Jumps in Finance. Springer, Heidelberg (2010)
35. Rose, G.: Kiss: A Bit Too Simple (2011). https://eprint.iacr.org/2011/007.pdf