DOI: https://doi.org/10.53052/9788367652001.12

Yurii SHCHERBYNA[1], Nadiia KAZAKOVA[2], Oleksii FRAZE-FRAZENKO[3]

WYKORZYSTANIE GENERATORA XORSHIFT DO SYMULACJI PROCESÓW STOCHASTYCZNYCH

Streszczenie: Przeprowadzono ocenę liniowego generatora kongruentnego oraz generatora "Mersenne Twister" i wykazano, że każda nierównomierność liczb na wyjściu generatora wybranego jako źródło losowości znacząco wpływa na jakość modelowanego procesu. Zbadano ekonomiczny z punktu widzenia zasobów obliczeniowych generator typu Xorshift i zaproponowano metodę rozrzedzania danych wejściowych w stosunku do numerycznego modelu przepływu poprzez usuwanie elementów, które nie pasują do rozkładu jednostajnego. Opisano kryterium odrzucania takich elementów.

Słowa kluczowe: modelowanie, liniowy generator kongruentny, generator Mersenne Twister, generator Xorshift, metoda funkcji odwrotnych, test chi kwadrat Pearsona, postprocessing przepływu numerycznego.

USING THE XORSHIFT GENERATOR TO SIMULATE STOCHASTIC PROCESSES

Summary: The evaluation of the linear congruent generator and the "Mersenne Twister" generator was performed and it was shown that any unevenness of the numbers at the output of the generator selected as a source of randomness significantly affects the quality of the process to be modeled. The Xorshift-type generator, which is economical from the point of view of computing resources, was studied and a method of thinning the input in relation to the numerical flow model by removing elements that do not fit into a uniform distribution was proposed. The criterion for rejecting such elements is described.

Keywords: Modeling, linear congruent generator, Mersenne Twister generator, Xorshift generator, inverse function method, Pearson's chi-square test, numerical flow post-processing.

1. Introduction

Recently, computer modeling has been experiencing rapid development. It is used not only at the preliminary stages of designing complex technological systems and

[1] Engineering Science Ph.D., National University "Odessa Law Academy", associate professor at the department of information technology, shcherbinayura53@gmail.com
[2] Prof. D.Sc., Odesa State Environmental University, Head of Department of information technology, kaz2003@ukr.net
[3] Engineering Science Ph.D., Odesa State Environmental University, associate professor at the department of information technology, frazenko@gmail.com

processes, but also allows for optimization and experimental evaluation of their constituent parts during tests.
The essence of modeling is to conduct a series of computational experiments, the purpose of which is scientific, analysis, interpretation and comparison of simulation results with the behavior of a real physical system or process. For this, software packages are created during modeling, which describe the behavior of systems and their parts, taking into account their interaction with each other and the external environment.
For the study of stochastic systems, statistical modeling is used, which involves multiple repetition of tests followed by processing of the obtained results. Usually, such methods involve the presence of pseudorandom number generators (PRNG) with a uniform distribution law on the interval $[0,\ 1]$ as the main source of the stochastic process. Moreover, the adequacy of the model to be implemented, regardless of the chosen modeling method, depends significantly on the degree to which the numerical flow at the output of the generator is uniform.
Digital replication of any pattern or process involving randomness requires that the chosen generation method produces sequences of numbers each of which can be reproduced repeatedly and meet a given uniformity criterion. Modeling experience shows that any unevenness significantly affects the quality of the process at the output of the computer model.
At the moment, there are a large number of methods for generating high-quality pseudorandom number generators, which include the MT generator, known as the Mersenne twister, MT, [2], Xorshift [3], linear congruent generator (LCG) [4] and many others. They output 32-bit or 64-bit numbers in [0, 2 32) and [0, 2 64) intervals. Unfortunately, all of them, without exception, do not pass the test for the uniformity of the probability distribution of the generated numbers and are not only unsuitable for cryptographic needs, but also for direct use in modeling [5]. Based on this, every time creating a computer model, developers should check the selected generator and, depending on the results, use additional methods of their randomization.
Unfortunately, uneven distribution of the output numerical stream is not the only drawback of relatively simple arithmetic generators of pseudo-random numbers. Another of their disadvantages is the consumption of a large number of computational operations during the generation process.
As shown in [5], the use of division operations and reduction of numbers by the appropriate modulo increases the amount of necessary computing resources, at least by an order of magnitude.
Despite the fact that the vast majority of known arithmetic algorithms were investigated and rejected as sources of randomness D. Knut [6] back in the last century, developers spare no efforts to improve them. Moreover, these efforts are simultaneously aimed at bringing the output numerical flow to a given level of uniformity of distribution, and at the most effective use of computing resources.
Most of the libraries of almost all known specialized software environments designed for solving research and engineering problems include such software generators as the MT generator, which has an extremely long repetition period ($2^{19937} - 1$ bit) and LCG. They both have both of the described disadvantages, and this especially applies to the MT generator. It is this disadvantage that makes them unsuitable for direct use for modeling purposes.

With this in mind, the goal of the study is to choose a simple and affordable generator that would be efficient in terms of computing resources and would provide the required uniformity of the output numerical flow.

2. Generator selection

As already mentioned, today is MT the default pseudorandom number generator is used by most C compilers, programming environments such as Python, the mathematical computing system Maple, and many others. Its supporters are primarily attracted by an extremely long repetition period, although this fact is not an indicator of quality and the implementation of such a generator requires large amounts of memory. Recent studies [7] show that it has serious shortcomings and should not be considered as a universal generator.

"Mersenne Twister" is the general name of a whole family of PRNG, the work of which is based on linear transformations over binary field $F^2 = \{0,1\}$. This means that the state of the generator is considered as n is a dimensional vector over the field F^2 and each subsequent state is a F^2-linear transformation. Since the sum in F^2 is just an operation XOR, such transformations are easily implemented and quickly calculated.

The main problem with using the MT generator for simulation purposes is that it fails statistical tests such as the Marsaglia binary rank test [8] and the linear complexity test [9].

The practice of using MT generators over the past 20 years has shown that its repetition period is $2^{19937} - 1$ of bits is too large. The implementation of such a generator, despite the simplicity of the operations underlying its algorithm, requires large amounts of processor cache memory, and this, in turn, reduces its performance. Even in situations where the initial number is chosen from the set y 2^{256} of possible options, the chance to get identical sequences is almost zero and the use of a generator that wastes a huge number of bits of processor cache memory does not make sense.

It is clear that an efficient model requires a simple, fast generator that does not consume an excessive amount of resources. At one time, George Marsaglia [10] gave mathematical justification of most of those described by Donald Knuth [6], iterative generators, which, after appropriate refinement, could be used as a source of randomness in simulation. In particular, he showed that an iterative generator requires the set of numbers to be the Z,inverse of a function f over a set Z, and uniform random selection of the initial number $z_0 \in Z$. Each subsequent number at the output of the iterative generator is formed according to the principle

$$f(z), f^2(z), f^{32}(z), \ldots, \qquad (1)$$

where $f^2(z)$ means $f(f(z)$, $f^3(z)$ means $f(f^2(z))$ etc. Usually, plural Z is the set of all possible 32-bit numbers, which represent m - tuples $x_1, x_2, \ldots, x_m$, and f is a function that converts the current tuple into the next tuple.

If f is a mutually unique function over Z, then for any initial number z, uniformly selected from Z, a random variable $f(z)$, will also be a uniformly distributed quantity. In case of random selection of a number $z$from Z, through the transformation $f(z), f^2(z), \ldots$ the formation of a sequence of homogeneous elections with Z. These

choices will not be random, but when used for simulation purposes, they behave and appear to be random.
The most famous iterative generator included in most software environments is the LKG generator. Each subsequent number at its output is formed from the previous number according to the principle

$$x_n = ax_{n+1} + k \; mod \; m \qquad (2)$$

This is the most well-known method of obtaining pseudorandom numbers, which requires the definition of parameters such as modulus m, an additive constant k and a random initial numeric x_0. If a the primary root of the number space p, a $x_0$is a random initial number from the set

$$Z = \{1, 2, \dots, p - 1\}, \qquad (3)$$

then the numerical sequence created according to principle (2) will be strictly periodic with a period $p - 1$ and each element of this sequence will be a pseudo-random uniformly distributed value in multiples Z.
The problem is that getting the value $ax \; mod \; p$ for a prime number p, is usually much more difficult than obtaining a value $ax \; mod \; 2^{32}$, because in the latter case, for most processors, it is performed automatically.
Thus, it can be argued that the LKG repetition period of the generator is equal to 2^{32}, a x_0 numbers belong to the plural $Z = \{1, 2, \dots, 2^{32} - 1\}$.
In search of an economical, from the point of view of computational resources, an PVC generator, George Marsaglia developed an algorithm known as Xorshift [11]. Today, there are a large number of its modifications, but in general, such a generator is a number of linear feedback registers (LFSR), which provide special efficiency without using excessively sparse polynomials.
The theory of Xorshift generators is based on the use 32 - or 64 -bit integer as an element of the vector space in the binary field modulo 2. The composition of such vectors is performed through the xor operation. Together with the shift operation, this allows you to implement the necessary linear transformations in the vector space using a minimum amount of computing resources.
Xorshift Algorithm considers the set of all nonzero binary vectors 1×32 on Z, a f as a linear transformation over Z, represented by a non-degenerate binary matrix T size 32×32. In this case, for a random number $y \in Z$ the sequence at the generator output is described as $yT, \; yT^2, \; yT^3, \dots$, if and only if the order $T$is even $2^{32} - 1$ in the group of non-degenerate binary matrices of size $32 \times 32$and the sequence has a period $2^{32} - 1$.
Marsaglia showed that there is a simple and fast way of forming the matrix product yT can be implemented if the order

$$T = (I + L^a)(I + R^b)(I + L^c), \qquad (4)$$

where L is the matrix that affects the left shift by one. In C, this operation looks like $y \wedge = (y \ll 1)$. Accordingly, the matrix yL^aimplements the shift $y \wedge = (y \ll a)$. Because matrix $R$is a transposed matrix L, its use implements a right shift by one $y \wedge = (y \gg 1)$. This means that (4), for a random 32-bit number from Z, makes it possible to get each subsequent number in the sequence$yT, \; yT^2, \; yT^3, \dots$ For example, in the C language it might look like this

$$y\ \hat{}=y \ll 13\ ;\ y\ \hat{}=y \ll 17\ ;\ y\ \hat{}=y < 5 \qquad (5)$$

In work [10] it is shown that there are no 32- or 64-bit vectors of type (4), with one or two shifts, that have a full period. To obtain the maximum possible period, matrices are needed that implement three types of shifts. Marsaglia showed that there are 81 triples of numbers $[a, b, c]$,$a < c$ for which a binary matrix of type (4) has a period $2^{32} - 1]$. As with all LFSRs, the Xorshift parameters of the generator should be chosen as carefully as possible.
Sequences of this type with combinations $[a, b, c]$, given in the works [10,11] are the best in terms of speed and minimal computer system resources. Such generators pass the BigCrush test from the TestU 01 package, but their lower bits fail the linearity test.
Thus, the Xorshift generator proposed by George Marsaglia looks most suitable for use in non-cryptographic projects and, in particular, in the simulation of stochastic processes.

3. Randomization of the numerical flow from the Xorshift generator

The term "chance" refers to the uncertainty of an event that may occur in the future. From this point of view, arithmetic pseudo-random number generators generate numerical streams that appear to the outside observer to be random or almost random. But for the needs of modeling one randomness is not enough, it is necessary that the stream of numbers at the output of the PRN generator is evenly distributed, but this requirement is not sufficiently provided by any of the known generators. The fact is that each subsequent number at the output of the iterative recurrent generator is the result of performing mathematical operations on one or several previous numbers. Generator type and initial conditions provide only the period of the number flow, but not the uniformity of the distribution of numbers within the period. The problem is that, regardless of the method of forming a model of a particular stochastic process, the unevenness of the numbers at the output of the generator chosen as a source of randomness is completely transferred to the process that is the goal of modeling.
Today, the efforts of mathematicians are focused on finding ways to improve RNGs intended for the purposes of cryptography. As for modeling, it is still considered not the most basic stage of design and, therefore, a coincidence within 5÷10 percent with the theoretically expected value of the stochastic process at the output of the model is considered sufficient, but it is also not provided by generators that output a non-binary stream of integers or real numbers. For example, the Xorshift128 generator developed by G. Marsaglia, with the length of the original sequence of 1000 positive random 4-byte numbers, gives the 16-segment histogram shown in Figure 1.
As the tests show, the indicator χ^2- Pearson, calculated taking into account 15% accuracy, usually significantly exceeds the critical permissible value.
In many sources and in the works of G. Marsaglia himself [10], it is indicated that the lower digits of the numbers generated by the generator are less "random" than the numbers in their higher digits. Because of this, it is suggested to discard the lower digits of the original numbers or to exchange them for the higher digits of other numbers generated by the same generator. G. Marsaglia himself solved this problem

by using logical operations and operations landslide This allowed him to create one of the most economical and fast-acting but cryptographically unstable generators.

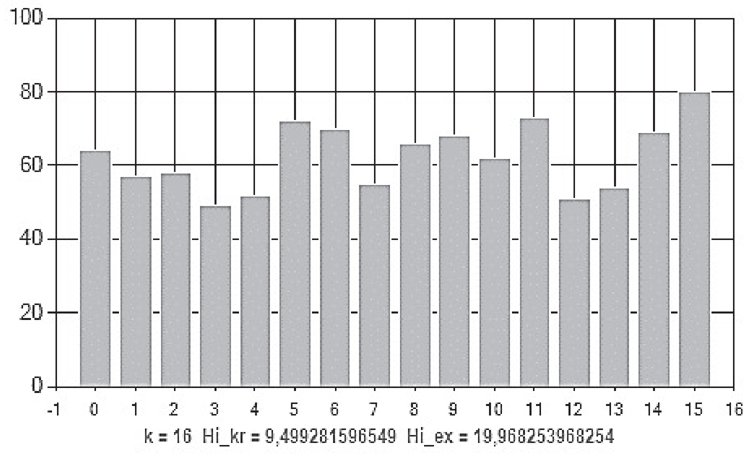


Figure 1. Histogram of the distribution of PVCs obtained using the Xorshift function 128

Figure 2 shows a diagram that explains the principle of such transformations for Xorshift128.

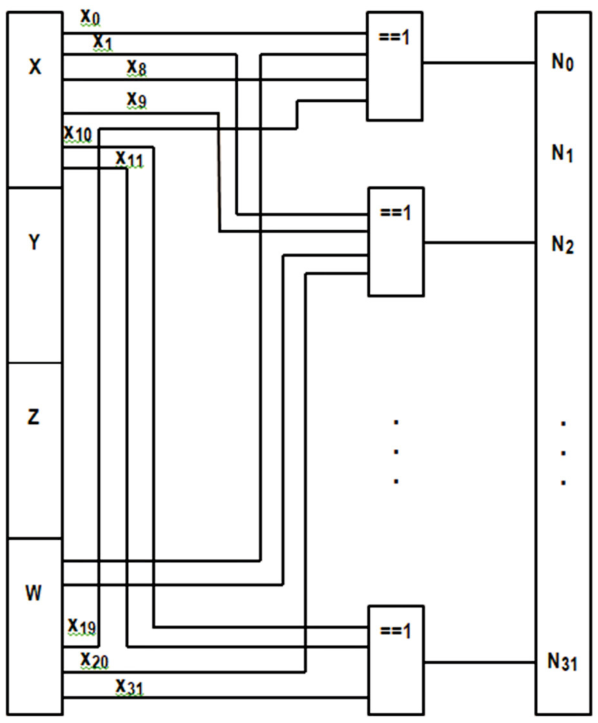


Figure 2. Scheme of the algorithm of the Xorshift 128 generator

The program fragment that implements such an algorithm has the following form.

```
x = 123456789
y = 362436069
z = 521288629
w = 88675123

t = x ^ ((x << 11) & 0xFFFFFFFF);
x = y;
y = z;
z = w;
w = (w ^ (w >> 19)) ^ (t ^ (t >> 8));
```

When starting the generator, four initial numbers are set $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$ and $\boldsymbol{w}$. They determine the internal state of the generator. Each subsequent number $\boldsymbol{w} = \{\boldsymbol{N_0}, \boldsymbol{N_1}, \ldots, \boldsymbol{N_{31}}\}$ is formed as a combination of digits $\boldsymbol{x}$and a previous value $\boldsymbol{w}$as shown in the code snippet, followed by a number shift $\boldsymbol{y} \to \boldsymbol{x}$, $\boldsymbol{z} \to \boldsymbol{y}$ and $\boldsymbol{w} \to \boldsymbol{z}$_ According to Marsaglia, it is precisely these shifts that increase the "randomness" of lower order numbers.

One controversial question arises here: do the lower order bits of a number really reduce the randomness of the numbers at the output of the generator? If we consider an ordinary counter, then the numbers in the lower digits of the numbers at its output are indeed repeated with a higher frequency, but the PRN generator is not a counter. This is the first. Second, there are generally few numbers that are limited to one byte or shorter. A generator of such numbers would have a limited period in size $2^n, n \leq 8$. It is within such limits that the "randomness" of the lower bits at the output of the generator should be evaluated. If the sample size is larger than the repetition period, the picture will not be objective.

For example, the numbers in the last byte at the output of the MT generator give the sequence distribution of 256 numbers shown in Figure 3(a).

For 256 numbers in the highest byte at the output of this sequence, the distribution pattern does not differ (Figure 3b).

The numbers shown on these diagrams whose horizontal coordinates coincide are the same numbers. The same is true for numbers for which the vertical lines coincide coordinates Repeated repetition and analysis of the diagrams shown in Figures 3a and 3b points to the dubiousness of the statement that the combinations of bits in the lower bytes are less random. In older bytes, the picture is the same.

This is especially noticeable if the size of the number is smaller than a byte. For example, in the same experiment, for $n = 5$, in the least significant byte, the distribution pattern looks as shown in Figure 3c. In the most significant byte, it looks like Figure 3d.

Thus, the probability of a number in a numerical stream should be considered in its entirety. It should also be remembered that for the needs of modeling, as a rule, real numbers are used and their lower digits are not deterministic.

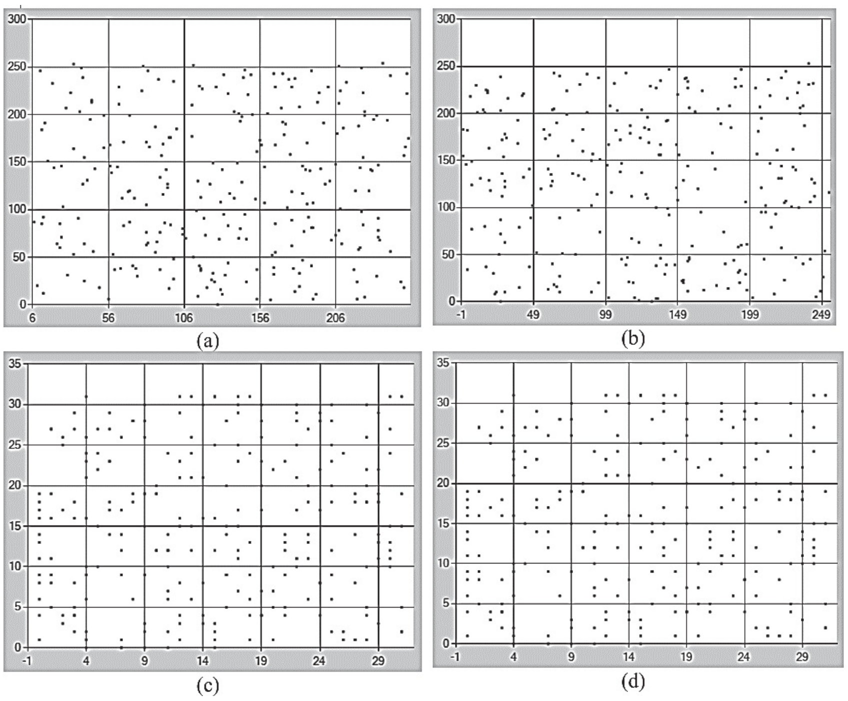


Figure 3. Distribution of 256 numbers at the output of the MT generator in the last byte

The positive effect of the algorithm embedded in Xorshift128 is that the use of logical bit operations and shift operations provides "whitening" and "shuffling" of the bits of numbers at the RNG output, as it is done in cryptographic generators. But, despite its positive qualities, this generator, like its other variants, does not provide greater uniformity of output numbers, compared to existing other non-cryptographic generators. This is clearly visible from the histogram shown in Figure 1. For most numerical samples at the output of the generator, the χ^2-Pearson indicator significantly exceeds the critical value.

As shown in [12], uniformity can be ensured by methods of post-processing of the numerical flow at the output of the generator. First of all, it means "rejecting" those numbers that do not fit into the uniform distribution. Usually, a criterion is chosen to assess uniformity χ^2- Pearson, because it is not tied to the type of distribution, and if a stream does not meet the requirements of this criterion, then, most likely, it will not satisfy the other criteria. The quality indicator of this criterion is determined by the rule

$$\chi^2 = \sum_{i=1}^{k} \frac{(n_i - N_i^*)^2}{N_i^*}, \tag{5}$$

where k is the number of histogram segments, $n_i$andN_i^* – the number of random numbers of the output stream that actually fell into the ith interval and their expected number, respectively. Expected number with uniform distribution N_i^* is defined as N/k.

The paper [13] gives recommendations on how to choose the optimal number of histogram intervals and the methodology for assessing the coincidence of the distribution of the numerical flow with its expected type. It is determined that when modeling a stochastic process, the parameters of which are known, the number of histogram intervals is not of great importance and should lie within 12÷18.

One of the methods of selecting numbers from the stream at the RNG output is described in [14]. Its essence is that, based on the type of distribution, the sample size N, and number of intervals k, the number of numbers falling into each interval is calculated. In the case of uniform distribution, these values should coincide and be equal N_i/k.

Mathematical expectation of the value $m_i$that should fall into each separate segment of the histogram $x_{min} \le x_i < x_{max}$, calculated as $m_i = (x_{min} + x_{max})/2$. In this case, the sum of the numbers S_i, which should fall into the ith interval, will be approximately equal to the value $S_i^* = N_i^* m_i$. If the sum of numbers really fell into the ith segment of the histogram S_i, then every time it will differ from the expected value S_i^*.

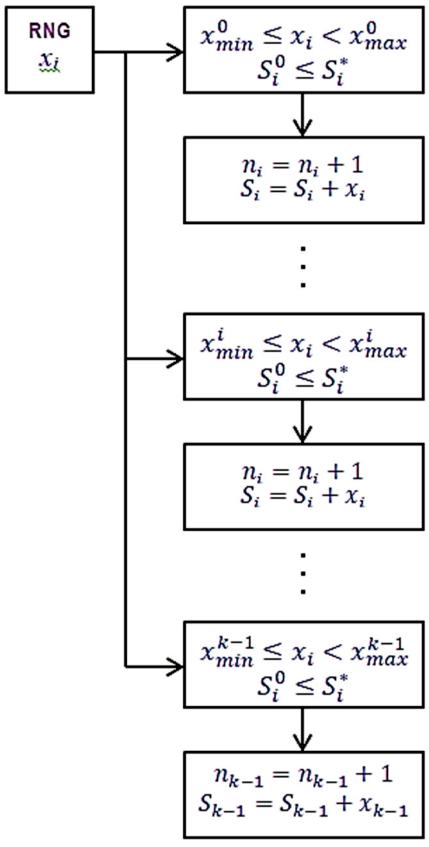
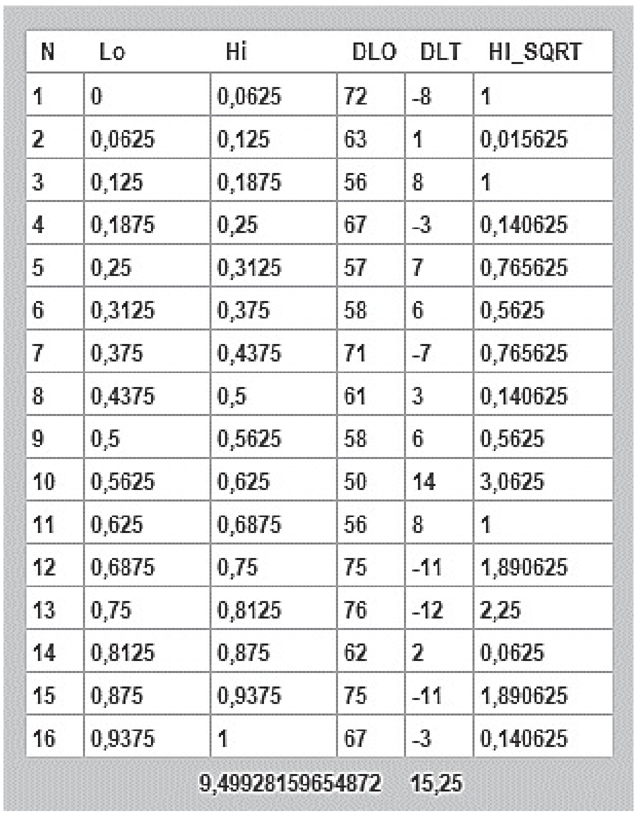


Figure 4. Randomization of numbers at the output of the Xorshift128 generator.

Thus, each time, after the generator generates the next number, it must be checked, firstly, in which segment of the histogram it fell and, secondly, how the sum of the

numbers in its segment has changed, as shown in Figure 4. If $S_i \leq S_i^*$, this number is added to the segment. Otherwise, the numbers falling into this segment are not taken into account. The peculiarity of this method is that the segments are filled not one at a time, but all in parallel, that is, only a small part of them is discarded as "extra". The conducted studies show that it does not exceed 2÷3 percent.
Results of tests of sampling of real numbers of length $N = 1024$, at the output of the Xorshift128 generator are shown in Figure 5.



N	Lo	Hi	DLO	DLT	HI_SQRT
1	0	0,0625	72	-8	1
2	0,0625	0,125	63	1	0,015625
3	0,125	0,1875	56	8	1
4	0,1875	0,25	67	-3	0,140625
5	0,25	0,3125	57	7	0,765625
6	0,3125	0,375	58	6	0,5625
7	0,375	0,4375	71	-7	0,765625
8	0,4375	0,5	61	3	0,140625
9	0,5	0,5625	58	6	0,5625
10	0,5625	0,625	50	14	3,0625
11	0,625	0,6875	56	8	1
12	0,6875	0,75	75	-11	1,890625
13	0,75	0,8125	76	-12	2,25
14	0,8125	0,875	62	2	0,0625
15	0,875	0,9375	75	-11	1,890625
16	0,9375	1	67	-3	0,140625

9,49928159654872 15,25

Figure 5. Distribution of numbers from the output of the Xorshift128 generator in histogram intervals without post-processing

If in this case the value of the accuracy indicator is chosen $\lambda = 0.15$ and the number of histogram intervals, which is equal to 16, is the critical value of the indicator χ^2- Pearson will be there $\chi^2 = 9.4292$, and the real one is $\chi^2_{\text{кр}} = 15.255$, that is, such a sequence will not be uniform.
Figure 7 shows the distribution of numbers of this sequence in histogram intervals, and the histogram itself is shown in Figure 6.

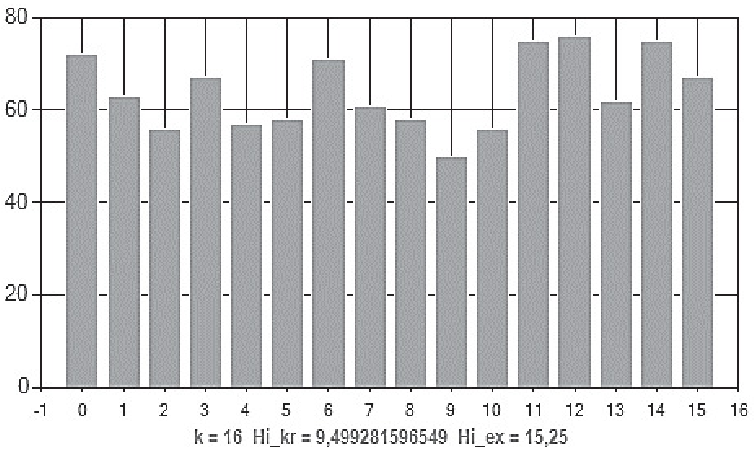
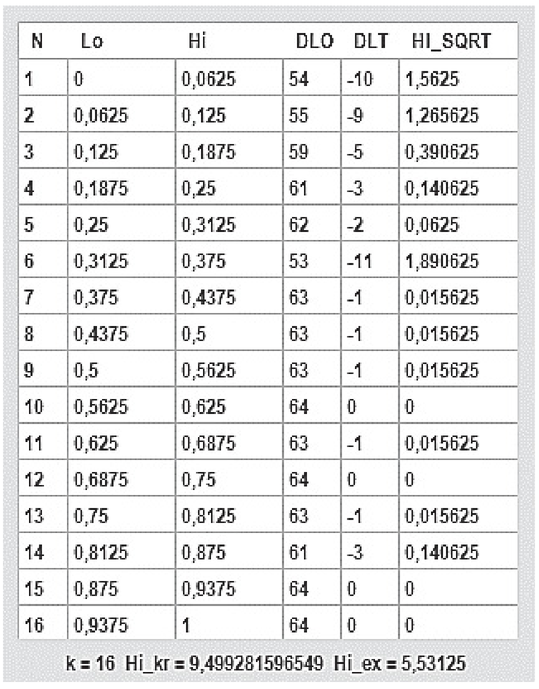


Figure 6. Histogram of the distribution of numbers from the output of the Xorshift128 generator

In case of applying the described method of post-processing, distribution of 1024 numbers from the output of the Xorshift128 generator, 48 numbers will be rejected, and 976 numbers will remain in the original sample. The distribution of these numbers in histogram intervals is shown in Figure 7, and the corresponding histogram is in Figure 8.



N	Lo	Hi	DLO	DLT	HI_SQRT
1	0	0,0625	54	-10	1,5625
2	0,0625	0,125	55	-9	1,265625
3	0,125	0,1875	59	-5	0,390625
4	0,1875	0,25	61	-3	0,140625
5	0,25	0,3125	62	-2	0,0625
6	0,3125	0,375	53	-11	1,890625
7	0,375	0,4375	63	-1	0,015625
8	0,4375	0,5	63	-1	0,015625
9	0,5	0,5625	63	-1	0,015625
10	0,5625	0,625	64	0	0
11	0,625	0,6875	63	-1	0,015625
12	0,6875	0,75	64	0	0
13	0,75	0,8125	63	-1	0,015625
14	0,8125	0,875	61	-3	0,140625
15	0,875	0,9375	64	0	0
16	0,9375	1	64	0	0

k = 16 Hi_kr = 9,499281596549 Hi_ex = 5,53125

Figure 7. Distribution of numbers from the output of the Xorshift128 generator in histogram intervals after post-processing

Multiple repetition of experiments shows that in the vast majority of cases the indicator $\chi^2 < \chi^2_{\text{кр}}$. This means that the additional processing of the numerical stream at the RNG output allows you to obtain a source of randomness suitable for modeling stochastic processes. Thus, " slitting " of the incoming flow from MT generator, gives significantly better simulation results from the point of view of their reliability.

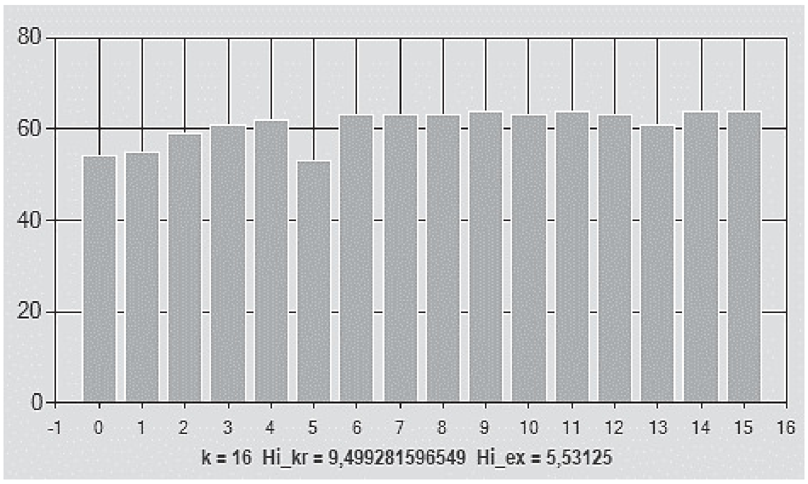


Figure 8. Histogram of the distribution of numbers from the output of the Xorshift128 generator after post-processing

4. Conclusions

From the results of the study of the vast majority of non-cryptographic generators of pseudo-random numbers, which are part of such software environments as Boost, Glib, C++, Python, Ruby, R, PHP, MATLAB and Autoit, it follows that they cannot be directly used as a source of randomness for modeling stochastic processes by the method of inverse functions or using the Monte Carlo method. In order for the numerical sequences at the output of such generators to appear truly random, the developers focused their efforts on providing them with as large a repetition period as possible. The authors of the MT-generator were especially successful in this. For testing such generators, the same test packages as for cryptographic generators are usually used. For example, such a package as [15]. That is, the uniformity of the numerical sequence is examined at the bit level. But the bit sequence divided into bytes and converted into the format of integers or real numbers does not preserve the uniformity of distribution, and for the needs of modeling, exactly such, uniformly distributed, numbers are needed.

The elimination of the problem of the unevenness of the numerical flow from recurrent generators was proposed from the very beginning to be solved by additional processing methods, but all works in this area, again, are focused on cryptographic generators, the requirements for which are orders of magnitude higher than for generators used for simulation. In the latter case, it is sufficient that the numerical sequence successfully passes the χ^2-Pearson test.

In addition to unevenness, one problem is the excessive amount of computing resources required by the operation of the MT generator offered in most software environments. A good way to eliminate this problem is to use the one suggested and described by George Marsaglia of the economic Xorshift generator and "sieving" the numbers at its output in the proposed effective way, which involves focusing on the mathematical expectation of numbers hitting each interval of the histogram.

REFERENCES

1. LAW A. M.: Simulation modeling and analysis, 5th. ed., McGraw-Hill Education, 2 Penn Plaza, New York, 2015. URL: *https://industri.fatek.unpatti.ac.id/wp-content/uploads/2019/03/108-Simulation-Modeling-and -Analysis-Averill-M.-Law-Edisi-5-2014.pdf*
2. MAKOTO MATSUMOTO, TAKUJI NISHIMURA: Mersenne Twister : A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. ACM Trans. Model. Comput. Simul. 8, 1998, 3–30. https://doi.org/10.1145/272991.272995 *URL: https://dl.acm.org/doi/pdf/10.1145/272991.272995*
3. PANNETON F., L'ECUYER P.: On the Xorshift Random Number Generators. ACM Trans. Model. Comput. Simul. 15, 2005, 346–361. https://doi.org/10.1145/1113316.1113319. URL: *https://web.archive.org/web/20210126143346/http://www.iro.umontreal.ca/~lecuyer/myftp/papers/xorshift.pdf*
4. NIEDERREITER H.: Quasi-Monte Carlo methods and pseudo-random numbers. doi: https://doi.org/10.1090/S0002-9904-1978-14532-7. URL: *https://www.ams.org/journals/bull/1978-84-06/S0002-9904-1978-14532-7/S0002-9904-1978-14532-7.pdf.*
5. LEMIRE D.: Fast Random Integer Generation in an Interval. doi: https://doi.org/10.1090/S0002-9904-1978-14532-7. URL: *https://arxiv.org/pdf/1805.10941.pdf*
6. DE KNUTH: The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd. ed., Boston, Mass, USA : Addison-Wesley, Longman Publishing, Addison-Wesley, Reading, Mass, 1998. URL: *https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/The%20Art%20of%20Computer%20Programming%20%28vol.%202_%20Seminumerical%20Algorithms%29%20%283rd%20ed.%29%20%5BKnuth%201997-11-14%5D.pdf*
7. VIGNA S.:. It Is High Time We Let Go Of The Mersenne Twister. 2019. URL: *https://www.researchgate.net/publication/336576895_It_is_high_time_we_let_go_of_the_Mersenne_Twister.*
8. MARSAGLIA G., LIANG-HUEI TSAY: Matrices and the structure of random number sequences. Linear Algebra Appl. 67(1985), 147–156. https://doi.org/10.1016/0024-3795(85)90192-2 URL: *https://www.sciencedirect.com/science/article/pii/0024379585901922*
9. GLYN C.D.: A aspects of local linear complexity. Ph.D. Dissertation. University of London. 1989. URL: *https://repository.royalholloway.ac.uk/items/1d5398b7-ba02-4820-8b50-60052a0bf7ad/1/*

10. MARSAGLIA G.: Random Number Generators. 2003. DOI 10.22237/ jmasm /1051747320 URL: *https://digitalcommons.wayne.edu/jmasm/vol2/iss1/2/*
11. MARSAGLIA G.: Xorshift RNGs. 2003. DOI:10.18637/jss.v008.i14. URL: *https://www.researchgate.net/publication/5142825_Xorshift_RNGs*
12. VON NEUMANN J.: Various techniques for use in connection with random digits. Applied Math Series, Notes by GE Forsythe, in National Bureau of Standards, 12(1951), 36 – 38, URL: *https://mcnp.lanl.gov/pdf_files/nbs_vonneumann.pdf.*
13. KNUTH K.: Optimal Data-Based Binning for Histograms. University at Albany (SUNY). Albany NY 12222, USA. 2013. URL: *https://www.academia.edu/50084430/Optimal_data_based_binning_for_histograms*
14. SHCHERBYNA Y., KAZAKOVA N., FRAZE-FRAZENKO O.: The Mersenne Twister Output Stream Postprocessing. CEUR Workshop Proceedings, 2021, 3200, pp. 265–273, URL: *http://star.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-3200/paper39.pdf*
15. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. SP 800-22 Rev. 1a, April 2010. URL: *https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf.*