# PSEUDO-RANDOM NUMBER GENERATOR BASED ON NEURAL NETWORK

**Migran N. Gevorkyan[1, a], Anastasia V. Demidova[1, b], Anna V. Korolkova[1, c], Dmitry S. Kulyabov[1,2, d], Leonid A. Sevastianov [1,3, e], Ivan M. Gostev [4, f]**

[1] *Department of Applied Probability and Informatics, Peoples' Friendship University of Russia (RUDN University), 6 Miklukho-Maklaya St, Moscow, 117198, Russian Federation*

[2] *Laboratory of Information Technologie, 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*

[4] *Department of Information Systems and Digital Infrastructure Management National Research University Higher School of Economics 20 Myasnitskaya str., Moscow 101000, Russia*

E-mail: [a] gevorkyan_mn@rudn.ru, [b] demidova_av@rudn.ru, [c] korolkova_av@rudn.ru, [d] kulyabov_ds@rudn.ru, [e] sevastianov_la@rudn.ru, [f] igostev@hse.ru

In this paper we consider pseudo-random uniformly distributed number generators, such as `xorshift` and `KISS`. We also provide `C++` implementation source code snippets of these algorithms and the results of tests made with dieharder utility. We also mention the possibility of parallelization and briefly discuss the idea of neural networks usage for pseudo-random numbers generation.

Keywords: pseudo-random numbers, pseudo-random numbers generation, neural networks, algorithm parallelization.

# 1. Introduction

In this paper, we consider algorithms for generating uniformly distributed pseudo-random numbers by using bitwise operations. Two classes of such algorithms: `xorshift` [1,2] and `kiss` [3] are particularly simple, but give a high quality sequence of pseudo-random numbers. To verify the quality of the numbers sequence, we use test suite `dieharder` [4]. Also in the conclusion we discuss the idea of using neural networks to generate a sequence of pseudo-random numbers.

The second volume of Donald Knuth book [5] gives an overview of generators of uniformly distributed pseudo-random numbers and criteria for assessing their quality. The author puts forward the idea, that the complexity of the generation algorithm is not related to the quality of the resulting sequence. The algorithms we consider in this paper clearly illustrates this idea.

# 2. The Kiss Generator

A family of generators, which gives high-quality sequence of pseudo-random numbers [3] got `KISS` acronym (Keep It Simple Stupid) because of the small number of steps and simpleness of operations. The `KISS` algorithm is used in `therandom_number()` procedure of the Frotran language (gfortran compiler [6])

The following C++ source code implements two versions of this generator: `KISS` and `jKISS`

```
unsigned long long int kiss(std::vector<unsigned long long int>& seed) {
	unsigned long long int t;
	const unsigned long long int a = 698769069llu;
	seed[0] = 69069 * seed[0] + 123456;
	seed[1] = seed[1] ^ (seed[1] << 13);
	seed[1] = seed[1] ^ (seed[1] >> 17);
	seed[1] = seed[1] ^ (seed[1] << 5);
	t = a * seed[2] + seed[3];
	seed[3] = (t >> 32); seed[2] = t;
	return  seed[0] + seed[1] + seed[2];
}
unsigned long long int jkiss(std::vector<unsigned long long int>& seed) {
	unsigned long long int t;
	seed[0] = 314527869 * seed[0] + 1234567;
	seed[1] = seed[1] ^ (seed[1] <<      5);
	seed[1] = seed[1] ^ (seed[1] >>      7);
	seed[1] = seed[1] ^ (seed[1] << 22);
	t = 4294584393llu * seed[2] + seed[3]; seed[3] = t >> 32;
	seed[2] = t;
	return  seed[0] + seed[1] + seed[2];
}
```

## 3. The xorshift Generator

The family of `xorshift` generators were developed in 2003 by John. Marsala (G. Marsaglia) [1, 2]. These generators are named after two bitwise operations: exclusive or (xor) and left and right bitwise logical shift. In addition to these two operations only usual arithmetic operations, such as of addition and multiplication, are used.

The following C++ source code implements the three variants of the `xorshift` algorithm.

```
unsigned long long int  xorshift64star(unsigned long long int seed) {
	unsigned long long int x; x = seed;
	x = x ^ (x >> 12);
	x = x ^ (x << 25);
	x = x ^ (x >> 27);
	return x * 2685821657736338717ull;
}
unsigned long long int xorshift128plus(std::vector<unsigned long long int>& seed){
	unsigned long long int x = seed[0];
	const unsigned long long int y = seed[1];
	seed[0] = y;
	x = x ^ (x << 23);
	seed[1] =  x ^ y ^ (x >> 17) ^ (y >> 26);
	return seed[1] + y;
}
unsigned long long int  xorshift(std::vector<unsigned long long int>& seed) {
	unsigned long long int t; t = seed[0];
	t = t ^ (t << 11); t = t ^ (t >> 8);
	seed[0] = seed[1];
	seed[1] = seed[2];
	seed[2] = seed[3];
	seed[3] = seed[3] ^ (seed[3] >> 19);
	seed[3] = seed[3] ^ t;
	return seed[3];
}
```

## 4. The xorshift Generator

The quality of the generated sequence of pseudo-random uniformly distributed numbers is as higher, as this discrete sequence properties correspond to the properties of the theoretical uniform continuous distribution. The summary criteria which we can use to assess the degree of compliance can be found in the book [5] and in article [7].

One of the most complete software implementations of the various tests for assessing the quality of the obtained pseudo-random sequences is the set of tests dieharder [4], which is implemented as a command-line utility. This utility is well supported and regularly updated.

We tested the generators by storing the generated sequence in a text file as a single column of pseudo-random numbers. The dieharder utility imposes requirements on the text file format. Each number must be on a new line, and the first lines of the file must contain the following data: the type of numbers (`d` — double-precision integers), the quantity of numbers in the file, and the bit width of the numbers (32 or 64 bits). Here is the example of the beginning of such a file:

```
type: d
count:  5
numbit: 64
1343742658553450546
16329942027498366702
.................
```

When such a file is created, one can pass it to dieharder for testing purposes:

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

where the `-a` flag turns on all built-in tests and the `-f` flag specifies the file to analyze. The test results will be saved to `file.out`. For a complete test, you need to generate about $10^9$ numbers (the exact number varies from test to test). In the case of fewer numbers, the test results may be worse than the theoretical capabilities of the algorithm.

## 5. Test Results and Conclusions

All generators described in the first part were implemented in C++ language and tested with the help of dieharder utility. The test results are summarized in the table 1.

Table 1. Results of DieHarder's tests

| Generator | Failed | Weak | Passed |
|---|---|---|---|
| KISS | 0 | 3 | 110 |
| jKISS | 0 | 4 | 109 |
| XorShift | 0 | 4 | 109 |
| XorShift+ | 0 | 2 | 111 |
| XorShift* | 0 | 2 | 111 |
| MT (Mersenne twister) | 0 | 2 | 111 |
| dev/urandom | 0 | 2 | 111 |

In addition to the algorithms `KISS` and `xorshift` we tested a well-known algorithm called Mersenne twister [8] and the pseudo-device urandom which is available in the operating systems of the Unix family.

The results show that the algorithms `xorshift*`, `xorshift+` and Mersenne twister, as well as the pseudo-device urandom, give the same qualitative sequence. The `xorshift*` and `xorshift+` algorithms, however, are much more simple than Mersenne twister and are independent of the [9] operating system unlike urandom device.

## 6. Notes to the Parallel Version

The considered generators can be efficiently parallelized to almost any number of threads or processes. If one wants to store all the generated numbers in RAM, one may run out of memory. Otherwise, the memory consumption will be minimal and it will be required to save only constants and from 1 or 4 last generated random numbers as seeds for next iteration.

It is also necessary to provide different seeds for each thread or process, otherwise they will all generate the same sequence of pseudo-random numbers.

## 7. Neural Networks Usage for Random Numbers Generation

In conclusion, we discuss the idea of using neural networks to generate random numbers. In the introduction the idea, expressed by D. Knuth, was pointed out that the quality of the generator does not depend on the complexity of the algorithm. After we have considered a number of modern algorithms and conducted comparative tests, this idea can be empirically supported by an example of the xorshift generator, which despite its simplicity showed results at the level of a much more complex Mersenne twister.

The `xorshift` algorithms use only low-level operations and do not require any high-level abstract data structures for their operation. The implementation we have given in C++ can be further simplified if we use arrays in the C-style instead of the vector container class.

This low level allows to achieve high performance and minimum memory consumption. Any implementation of the pseudo-random number generator using neural networks will be a priori less productive, as it will require significantly more resources to initialize the structure of neurons, as well as time for training. Even if the generator on neural networks will give a better sequence (which is difficult to achieve, since these algorithms perform weak only in 2 or 4 from more than a hundred tests), this advantage is likely to be completely neglected by a large consumption of computing resources.

## 8. Conclusion

The generators considered by us can be effectively implemented in compiled programming languages. They also scale efficiently to any number of threads or processes, allowing them to be used in conjunction with the Monte Carlo method. We also made some skeptical arguments about the idea of using neural networks to generate pseudo-random numbers.

## Acknowledgment

The publication has been prepared with the support of the “RUDN University Program 5-100” and funded by Russian Foundation for Basic Research (RFBR) according to the research project No 16-07-00556.

## References

[1] 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

[2] 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

[3] G. Rose, Kiss: A bit too simple (2011). URL https://eprint.iacr.org/2011/007.pdf

[4] R. G. Brown, D. Eddelbuettel, D. Bauer, Dieharder: A Random Number Test Suite (2018). URL http://www.phy.duke.edu/~rgb/General/rand_rate.php

[5] 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.

[6] The gfortran team, Using GNU Fortran (2015). URL https://gcc.gnu.org/onlinedocs/

[7] 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

[8] 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

[9] D. Jones, Good practice in (pseudo) random number generation for bioinformatics applications (May 2010). URL http://www0.cs.ucl.ac.uk/staff/D.Jones/GoodPracticeRNG.pdf