Symulacja w Badaniach i Rozwoju
Vol. 9, No. 1-2/2018

Joanna WIŚNIEWSKA
Wojskowa Akademia Techniczna
ul. gen. W. Urbanowicza 2, 00-908 Warszawa
E-mail: joanna.wisniewska@wat.edu.pl

Comparing quality of pseudo- and true-random numbers obtained from different sources

1 Introduction

Computer simulation often needs many random values to properly carry out the computation. For many years these values were delivered by Pseudo Random Number Generators (PRNGs). The practice made PRNGs [1, 3] useful tools: easy to implement, quick, producing numbers which does not seem to be pseudorandom. Advancements in quantum physics and quantum computing, however, may focus our attention on the problem of random and pseudorandom values. It is a fact that the quantum effects are sources of true random values and the availability of True Random Number Generators (TRNGs) increased lately (especially because of the prices). The second issue is simulating the behavior of a quantum system which seems to be a bit self-contradictory without TRNGs. Of course, we can always use built-in functions like RdRand which appeared in 2012 in Intel processors. The problem is that we should be aware of what the high quality of the random/pseudorandom numbers is.

In this work the sources of values, which may be used e.g. during computer simulation, are presented. Then the problem of testing values obtained from the generators is addressed. Finally, the results of some tests are presented.

2 Sources of values

At present, the numbers that are needed during the simulation may come from at least three different types of sources: PRNG, TRNG and Digital Random Number Generator (DRNG).

PRNGs are just algorithms which produce some values. The results produced by PRNGs are perfectly predictable. Because for many years there was no cheap alternative to produce needed numbers, many techniques were developed to obtain a very high quality of the generated values. The first factor is a seed. The generator's seed is a number (or numbers) which is the initial value for the algorithm producing numbers. If we run the algorithm with the same seed we always obtain the same pseudorandom numbers. To make an impression that the numbers are really random the PRNG may extract the current time from the system clock and use it as the seed. This approach does not change the fact that if someone knew the time when algorithm started, he could calculate what numbers were generated. The second issue is a periodicity of the generator. Each PRNG is bounded by the number of values that pose unrepeatable series. Nowadays, we can use PRNGs that guarantee high quality

numbers, e.g. MT19937 (period 4,3 x 10^{6001}), MRG32K3A (period 3,1 x 10^{57}), LFSR113 (period 10^{34}), GM61 (period 5,3 x 10^{36}).

TRNGs are physical generators which use a quantum physics phenomenon to obtain truly random values. We can imagine the whole procedure as a measurement of a single photon's state. Two basic states of one photon may be expressed as a vertical or horizontal polarization – just like a single bit which may hold a value 0 or 1. Quantum states have this advantage that except of being equal to one of the basic states, they may be a mixture of basic states. Using Dirac notation, a single photon state we can express as $|\varphi\rangle$:

$$|\varphi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle\,. \tag{1}$$

Let us assume that $|0\rangle$ corresponds to the vertical polarization and $|1\rangle$ to the horizontal polarization. Complex numbers α_0 and α_1, called amplitudes, describe the probability if the state after the measurement is, respectively, $|0\rangle$ or $|1\rangle$, therefore: $|\alpha_0|^2 + |\alpha_1|^2 = 1$.

The generator produces quantum state:

$$\left|\varphi\right\rangle = \frac{1}{\sqrt{2}}\left|0\right\rangle + \frac{1}{\sqrt{2}}|1\rangle\,. \tag{2}$$

As we can see both basic states are equally probable. Now, the state has to be measured. For example, the single photon, described as in (2), meets a vertical polarizer. If the photon goes through the polarizer then it is assumed that its state is $|0\rangle$. If the polarizer stops, the photon its state is $|1\rangle$. This simplified procedure shows how the TRNG produces a random number. Of course, if the 32-bit numbers are needed the procedure has to repeated 32 times to produce one number.

According to Moore's law the number of transistors in a dense integrated circuit doubles about every two years. About year 2010 engineers constructing and building processors noticed that sizes of transistors became so small that the stability of integrated circuits work is threaded by some quantum effects. In 2011 Intel used this fact and produced line of processors, called Ivy Bridge (22 nm die shrink), which allowed to generate true random values. A part of the circuit is not isolated from the external influence (mainly temperature changes), therefore the movements of holes/electrons are unpredictable there. When the measurement is made we randomly obtain value 0 or 1. This type of generator is called DRNG. Intel DRNG [2] hardware implementation offers two functions: *RdRand* for generating numbers influenced by the external influence and *RdSeed* which generates random numbers similarly to *RdRand* but uses them as seeds for PRNGs.

3 Batteries of tests for numbers generators

Many years of using PRNGs caused the need of testing the generators quality. The first battery of test, called *Diehard*, was proposed by George Marsaglia in 1995 [5]. In 2001 National Institute of Standard and Technology (Gaithersburg, Maryland, USA) published *NIST* battery which was regularly updated (last version was released in 2010). In 2003 Robert G. Brown proposed *Dieharder* battery which was an extended version of *Diehard*. In 2007 Pierre L'Ecuyer and Richard Simard prepared a package *TestU01* [4] which was composed of eight batteries of tests: *SmallCrush*, *Crush*, *BigCrush*, *Rabbit*, *Alphabit*, *pseudoDiehard*, *NIST* and *FIPS_140_2* [7].

In next section of this work results of tests for some pseudo- and true-random numbers will be presented. A battery chosen for this task is *Dieharder*, because the software is still developed and updated. It is very easy to install and use on Linux-like operating systems.

The *Dieharder* battery contains the following tests from Marsaglia's *Diehard*:

- Birthday spacing – m days from n-day year are generated ($n \geq 2^{18}$). The spacings between drawed days should be asymptotically exponentially distributed.
- Overlapping permutations – permutations of five consecutive numbers are generated. This 120 series should occur in an analyzed set with an equal probability.
- Binary rank matrices – 40000 binary matrices sized 32×32 are generated. The rank of every matrix is calculated. Matrices with the rank greater than 28 are counted. Numbers of these matrices should be consistent with chi-squared test.
- Monkey tests – four tests (Bitstream, Overlapping Pairs Sparse Occupancy, Overlapping Quadruples Sparse Occupancy, DNA Test) based on the infinite monkey theorem saying that if there is enough time and monkeys equipped with typewriters they are able write all Shakespeare's plays.
- Count the 1s – the generated series are divided to n-element sequences. In every sequence the number of 1s is counted. The sequences are converted to letters, e.g. if there is zero 1s letter A is producted, for two 1s letter B, etc. The distribution of letters in the series is analyzed.
- Parking lot test – in 2-dimentional array 100×100. There are 12000 numbers generated (from range 1 to 10000). The number of values generated only once should follow the normal distribution.
- Random spheres tests – in 2- or 3-dimentional array (1000×1000 or $1000 \times 1000 \times 1000$) 8000 or 4000, respectively, cells are randomly chosen. These cells become centers of the spheres with maximum radiuses not overlapping their neighbors. The distribution for the number of smallest spheres should be exponential.
- Squeeze test – the number 2^{31} is multiplied by random floats from range (0,1) until the result is equal 1. The test is performed 100000 times. The number of generated floats should follow a certain distribution.
- Overlapping sums – long sequences of floats from range (0,1) are generated. Every 100 consecutive floats are added. The sums should be normally distributed.
- Runs tests – long sequences of floats from range (0,1) are generated. The length of subseries of ascending and descending values are counted. The obtained sums should follow a certain distribution.
- Craps tests – 200000 games of craps are simulated. The numbers of wins and throws per game should follow a certain distribution.

The tests added by Robert G. Brown are:

- Marsaglia-Tsang GCD – 10^7 32-bit unsigned integers are generated. For every pair of the numbers their greatest common divisor (GCD) is calculated with use of the Euclid's Method. The GCDs and number of algorithm's steps should follow a certain distribution.
- STS Monobit test – counts the 1s in a long sequence of 32-bit unsigned integers. The sum is compared to the expected value.
- STS Runs – the binary series are generated. Counted are "0 run" and "1 run" which should follow a certain distribution. "0 run" begins with 10 and ends with 01. "1 run" begins with 01 and ends with 10.
- STS Serial tests – for the parameter $n = \overline{1,16}$, 2^n n-bit sequences are generated. The number of the same sequences should occur with the same probability.
- STS Serial tests with overlapping – tests similar to above, but additionally a cyclic wrap is realized on the analyzed series.
- RGB Bit Distribution tests – n-element tuples of 0s and 1s are generated where $n = \overline{1,12}$. The numbers of the same elements on the same positions should be consistent with the chi-squared test.
- RGB Minimum Distance Tests – four hybrid tests, made as the generalizations of Random Spheres and Minimum Distance tests (Minimum Distance test is originally *Diehard* test in which distances between 8000 points in 2-dimensional array (10000×10000) should follow exponential distribution).
- RGB Permutations tests – $n!$ permutations of n numbers are generated ($n = \overline{2,5}$). In the tested series all permutations should occur the same number of times.
- RGB Lagged Sum tests – the generated numbers are added with skipping of n elements ($n = \overline{0,32}$). The average value of the summed numbers should fit to the average values of numbers from the generator's range.
- RGB Kolmogorov-Smirnov test – the generated values should follow the Anderson-Darling or Kuiper Kolmogorov-Smirnov test.
- DAB Byte Distribution test – there are $256 \times n$ counters, n independent bytes are extracted from each of k consecutive words. The counters are increased if the bytes repeat in the words. Values of counters should be consistent with the chi-squared test.
- DAB DCT Frequency Analysis test – the Discrete Cosine Transform (DCT) is performed on the output of the number generator. The finite sequences of the data points, in terms of a sum of cosine functions oscillating at different frequencies, should be consistent with the chi-squared test.
- DAB Fill Tree test – a binary tree of the fixed depth is filled with the words from the generator. When a word cannot be inserted into the tree, the counter of words in tree is saved – these values should be consistent with the chi-squared test.
- DAB Fill Tree 2 test – similar to previous test, but instead of the words the trees are filled with the bits.

- DAB Monobit 2 test – a block-monobit test which counts the 1s in blocks of generated n-bit unsigned numbers. The size of the block is 2^k where $k = \overline{0,n}$. The sum is compared to the expected value.

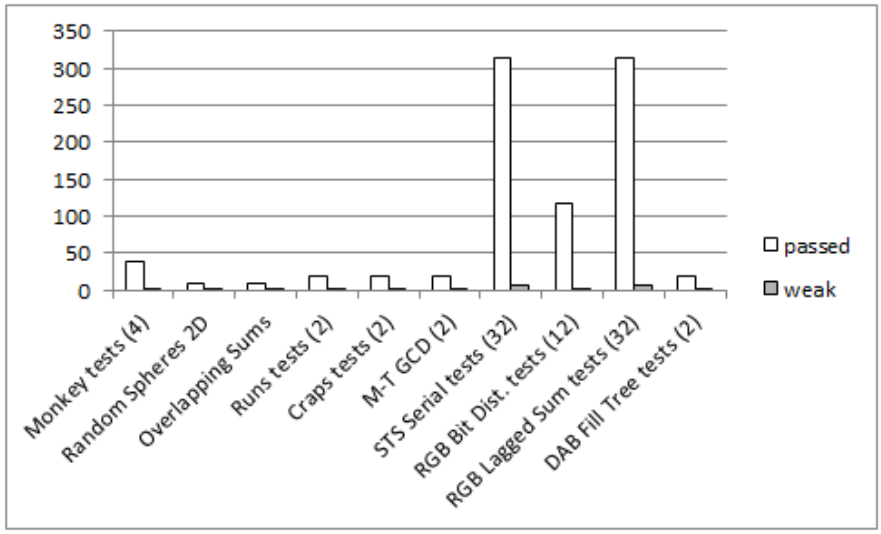
4 Tests results

Five different generators were tested with use of the *Dieharder* battery. For each generator the tests were run ten times. The total number of tests in the battery is 115. One single test may be evaluated as passed, weak or failed. If a single test, for the same generator, was ended ten times with a result "passed" then it is not shown at a graph. If there is a group of tests appearing under one name, the name of the test is ended with the number of subtests in brackets.

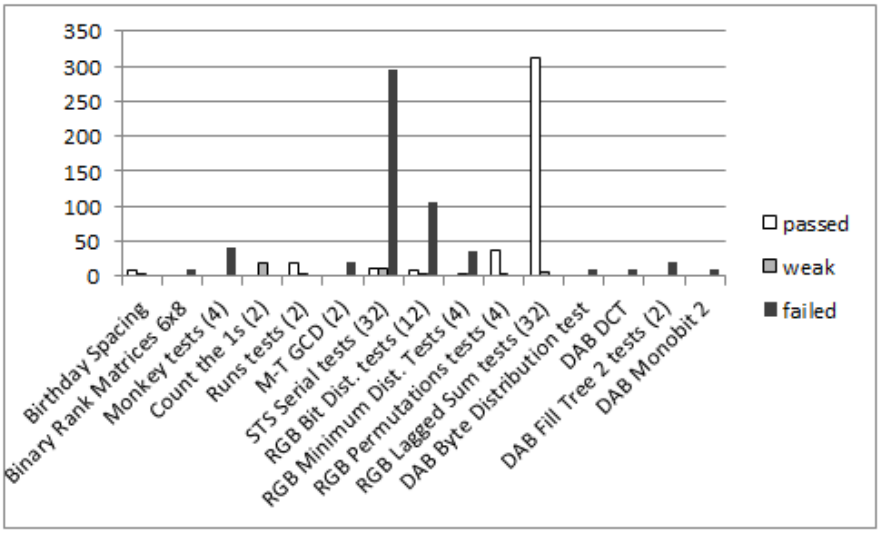
The first generator is PRNG termed as Mersenne Twister MT19937. It was elaborated by Makoto Matsumoto and Takuji Nishimura and generates 32-bit words [6]. The generator's period is 4,3 x 10^{6001}, therefore it is a very popular PRNG. The tests results are depicted in Fig. 1, the streams of numbers produced by MT19937 were directly transmitted to the *Dieharder* software.

To show the difference between one high-quality PRNG and another much poorer, the next tests were performed for the classic RAND generator used e.g. as a basic random function in the C programming language. The period of this generator is 2^{32}. The tests results are depicted in Figure 2 and streams of numbers were also directly transmitted to the *Dieharder* battery.

The third generator taken into account in this work is a TRNG. Unfortunately, there was no opportunity to have a direct access to this kind of device and the generated numbers could not be streamed to the *Dieharder* battery. However, the *Dieharder* software can test the files of numbers. According to the manual of *Dieharder* there should be at least 2.5 million 32-bit unsigned integers in a file to ascertain correct results of tests. Ten such files utilizing a TRNG, owned by Australian National University (ANU), were prepared. The TRNG streams the numbers and using ANU's website or special package for Python, the mentioned values can be saved into the files. The results of the test for these true random numbers are presented in Figure 3.



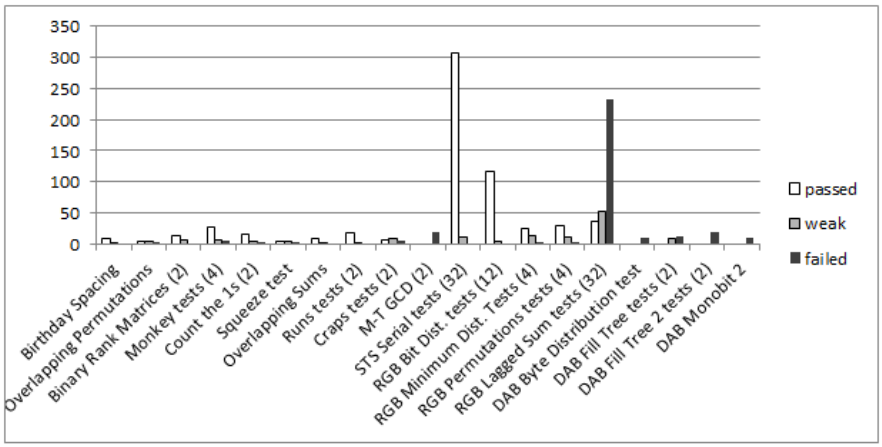
Rys. 1. Wyniki testów dla MT19937
Fig. 1. The tests results for the MT19937



Rys. 2. Wyniki testów dla funkcji Rand
Fig. 2. The tests results for the Rand function

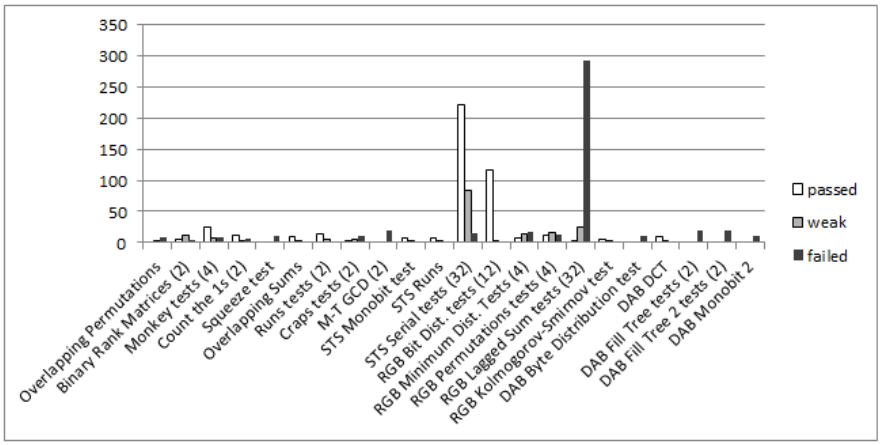
The fourth generator is Intel DRNG working on Windows 7 operating system. Because the *Dieharder* software is dedicated for Linux-like operating systems and numbers must be generated on Windows-like system with use of RdRand() function, also ten files

of values must be prepared (each 2.5 million of 32-bit unsigned integers). The tests for the Intel DRNG working on Windows 7 were run from these files. The tests results are depicted in Figure 4.



Rys. 3. Wyniki testów dla TRNG

Fig. 3. The tests results for the TRNG



Rys. 4. Wyniki testów dla RdRand w systemie Windows 7

Fig. 4. The tests results for the RdRand function on Windows 7

The last set of tests was performed for the RdRand() function on Linux Fedora 28. The tests results are depicted in Fig. 5 and the generated numbers were directly streamed to the *Dieharder* battery.

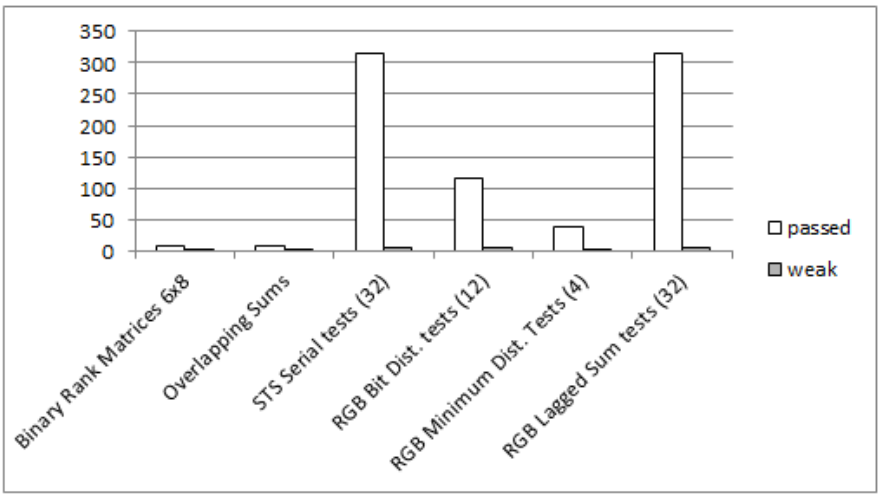
5 Summary

In this section the obtained results for five generators, with use of *Dieharder* battery of tests, will be commented. Additionally, it sounds reasonable to take into consideration also a correlation between the results of the tests for every generator.

Comparing results for the PRNGs: MT19937 and RAND; we can see that RAND failed in a greater number of the tests. In contrast to RAND, MT19937 did not obtain any "failed" mark. For both aforementioned generators the correlation between tests is very high. This means there are no significant differences between results of the tests, e.g. most of all tests for MT19937 were passed, RAND failed all monkey tests.

The results for the TRNG are not prefect what may be surprising. It obtained "weak" and "failed" marks in more tests than simple RAND generator. What can be observed for this generator is that there are some tests results which are not correlated: overlapping permutations were passed 4 times, failed twice and the mark "weak" was reached 4 times; squeeze test was passed 4 times, failed 3 times and the mark "weak" was obtained 3 times; in two craps tests there were 7 marks "passed", 8 marks "weak" and 5 marks "failed".

Taking into account the results for DRNGs, we can observe that the function RdRand differs a lot according to its implementation for the operating system. RdRand for Fedora behaves similarly to MT19937 and RdRand for Windows like the TRNG. RdRand on Fedora did not obtain any mark "failed" and the correlation between tests results is very high. RdRand on Windows has more varied marks and the correlation between tests results is lower, especially for: two binary rank matrices tests (6 times "passed", 11 times "weak", 3 times "failed"), four monkey tests (24 times "passed", 8 times "weak", 8 times "failed"), two craps tests (3 times "passed", 6 times "weak", 11 times "failed"), 32 STS serial tests (221 times "passed", 84 times "weak", 15 times "failed"), four RGB minimum distance tests (7 times "passed", 15 times "weak", 18 times "failed"), four RGB permutations tests (12 times "passed", 16 times "weak", 12 times "failed") and RGB Kolmogorov-Smirnov test (5 times "passed", 4 times "weak", 1 time "failed").



Rys. 5. Wyniki testów dla RdRand w systemie Fedora 28

Fig. 5. The tests results for the RdRand function on Fedora 28

Drawing conclusion from the above results seems difficult. The TRNG was supposed to be a benchmark in evaluation of numbers' quality. We can see that according to the *Dieharder* battery the TRNG is not a perfect generator. Anyway, if the true randomness is considered, we should remember that random sequence of five values may be: 54, 78, 300, 1, 83589 or 1, 1, 1, 1, 1. This was the reason to introduce the correlation between tests results: in four tests of the TRNG the correlation was low what may indicate true randomness. If the correlation is concerned, the DRNG for Windows obtains better characteristic that this implemented on Fedora.

The obtained results point some directions in evaluation of random numbers quality, but to gain convincing results much more tests should be performed.

Bibliography

1. Gentle J.E.: *Random Number Generation and Monte Carlo Methods* (Second edition). Springer-Verlag New York, 2003
2. Hamburg M., Kocher P., Marson M.E.: *Analysis of Intel's Ivy Brigde Digital Random Number Generator*. Raport Crytpography Research Inc., 2012
3. Herrero-Collantes M.: *Quantum Random Number Generators*. arXiv: 1604.03304v2 [quant-ph] 2016
4. L'Ecuyer P.: Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1): pp. 159-164, 1999
5. Marsaglia G.: A Current View of Random Number Generators, Computing Science and Statistics: *Proceedings of the 16th Symposium on the Interface*. Elsevier Science Publishers B.V., Amsterdam, 1985
6. Matsumoto M., Nishimura T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation* (TOMACS), 8(1): pp. 3-30, New York 1998

7. Rukhin A., Soto J., Nechvatal J., Smid M., Barker E., Leigh S., Levenson M., Vangel M., Banks D., Heckert A., Dray J., Vo S.: *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Raport NIST, 2010

Porównanie jakości liczb pseudolosowych i losowych pozyskanych z różnych źródeł

Streszczenie

Praca dotyczy problemu generowania liczb pseudolosowych. Wartości liczbowe wymagane podczas symulacji mogą być generowane przez: pseudolosowe generatory liczb, rzeczywiste generatory liczb losowych lub cyfrowe generatory liczb losowych. W artykule opisano zestaw testów, które pomagają w ocenie jakości uzyskiwanych wartości liczbowych. Dla pięciu różnych generatorów uruchomiono zestaw testów, a w pracy zamieszczono wyniki tych testów w postaci wykresów.

Słowa kluczowe: generator liczb (pseudo)losowych, jakość generatorów, symulacja

Summary

The manuscript refers to the problem of the pseudorandom numbers generation. Numbers needed during a simulation may be generated by pseudorandom numbers generators but also by true random numbers generators or digital random number generators. In this work some tests were described which help to evaluate quality of random values. For five generators, the batteries of tests were run and the manuscript contains results of these tests in a form of graphs.

Keywords: (pseudo)random numbers generators, quality of random values, simulation