Computing (2013) 95 (Suppl 1):S259–S275
DOI 10.1007/s00607-012-0267-z

Random number generation system improving simulations of stochastic models of neural cells

Karol Gugała · Aleksandra Świetlicka · Michał Burdajewicz · Andrzej Rybarczyk

Received: 30 September 2012 / Accepted: 11 December 2012 / Published online: 3 January 2013

Abstract The purpose of this work is to speed up simulations of neural tissues based on the stochastic version of the Hodgkin–Huxley model. Authors achieve that by introducing the system providing random values with desired distribution in simulation process. System consists of two parts. The first one is a high entropy fast parallel random number generator consisting of a hardware true random number generator and graphics processing unit implementation of pseudorandom generation algorithm. The second part of the system is Gaussian distribution approximation algorithm based on a set of generators of uniform distribution. Authors present hardware implementation details of the system, test results of the mentioned parts separately and of the whole system in neural cell simulation task.

Keywords Random number generator · Gaussian distribution · Uniform distribution · Hardware implementation · Hodgkin–Huxley model · Graphics processing unit · GPU

Mathematics Subject Classification 65C10 (Numerical analysis - Probabilistic methods, simulation and stochastic differential equations - Random number generation)

K. Gugała (✉) · A. Świetlicka · M. Burdajewicz · A. Rybarczyk
Chair of Computer Engineering, Faculty of Computing,
Poznan University of Technology,
Piotrowo 3a, 60-965 Poznan, Poland
e-mail: karol.gugala@put.poznan.pl

A. Świetlicka
e-mail: aleksandra.swietlicka@put.poznan.pl

M. Burdajewicz
e-mail: mburdajewicz@gmail.com

A. Rybarczyk
e-mail: andrzej.rybarczyk@put.poznan.pl

1 Introduction

Science since the very beginning tries to understand processes that take place in the nature. One of the methods to truly understand surrounding world is simulation. Unfortunately complexity of the natural processes entails an elongation of the simulations time and dramatically increase the need of computing power. This paper focuses on these problems, and propose solutions that speeds up simulations of natural processes and simplifies computations in some areas of it. We concentrate on simulations of neural cells based on the Hodgkin–Huxley model [6], especially on a stochastic version of it [9].

Simulations of biological processes (e.g. an electric potential flow on a neural cell membrane) often requires large amount of random values [5] with specified distribution. Quality of the projection of natural stochastic processes in simulation environment forces use of high entropy random generators. Increasing the generation speed of random values often leads to decrease its randomness. Building the high speed random generator without losing its level of randomness can be a milestone in the biological processes simulations.

Recently an often approach in simulation tasks is to use GPU implementation. The use of the well known very fast parallel pseudorandom generators becomes natural in this solution. Higher level of randomness demands the necessity of use of more complex, and hence slower, algorithms like Mersenne-Twister. Main disadvantage of the mentioned solutions is its possibility to generate only a pseudorandom values. In the natural processes, which we are trying to simulate, there is no such thing as pseudo randomness.

On the other hand there is a number of truly random hardware generators available on the market. These generators are based on such phenomena as temperature flow, space radiation, flashing of the pulsar, etc. The main disadvantage of these solutions is the generation speed, and often very slow random data transfer rate.

In this paper we introduce solution that combines advantages of mentioned families of the random generators.

Second aspect of this work is to concentrate on acquiring proper distribution of random data for neural cell simulation purposes. Stochastic version of the Hodgkin–Huxley neural cell model requires Gaussian distribution of random data. Classic approach uses Box–Muller transformation [Eq. (1)] to acquire eligible distribution. In this paper we are showing Gaussian distribution approximation algorithm basing on the set of uniform distribution values. Further in the paper we will show that our generator is faster than classic Box–Muller transform approach, and quality of approximated data is satisfactory in simulations based on the Hodgkin–Huxley neural cells model.

$$\begin{aligned} z_1 &= \sqrt{-2\ln u_1}\,\sin(2\pi u_2) \\ z_2 &= \sqrt{-2\ln u_1}\,\cos(2\pi u_2) \end{aligned} \tag{1}$$

In the next section we briefly describe the Hodgkin–Huxley neural cell model and stochastic version of it. After that, in Sect. 3, we present our proposal of a random generation system. Next, in Sect. 4, Gaussian approximation algorithm is presented.

In Sect. 5 sample implementation tests results are shown. Finally, in Sect. 6, discussion on results presented in section five and short summary of the paper are presented.

2 Hodgkin–Huxley neural cell model

Models, that we are considering, are describing the potential on the neuron's membrane, and are based on the idea of an equivalent electrical circuit which is described with a differential equation of the form [6]:

$$C\frac{dV}{dt} = I - g_{Na}m^3h(V - V_{Na}) - g_K n^4(V - V_K) - g_L(V - V_K) \tag{2}$$

where V is a potential of the neuron's membrane, g_{Na}, g_K and g_L are the conductivities of certain types of ions (sodium, potassium and chloride, respectively) that flow through the membrane. Parameters V_{Na}, V_K and V_L are representing the reversal potential of sodium, potassium and chloride ions, respectively, while I is an input current of the neuron. Additionally the Hodgkin–Huxley model is described with three following equations [6]:

$$\begin{aligned}
\frac{dn}{dt} &= \alpha_n(V)\cdot(1-n) - \beta_n(V)\cdot n \\
\frac{dm}{dt} &= \alpha_m(V)\cdot(1-m) - \beta_m(V)\cdot m \\
\frac{dh}{dt} &= \alpha_h(V)\cdot(1-h) - \beta_h(V)\cdot h
\end{aligned} \tag{3}$$

of variables m, n and h, which are showing the behaviour of the single channel, which is controlling the movement of ions in-between interior and exterior of the neuron. A single gate, in most of the basic models, can be in one of two states—opened (permissive) and closed (non-permissive), and the first part of each equation $\alpha_x(V)\cdot(1-x)$ refers to the transition from non-permissive to permissive state, while part $\beta_x(V)\cdot x$—conversely [6].

Furthermore in the model we used an assumption that a single channel can be at once in one of few states, where only one is permissive [3]. To describe the relationship between the states we used Markov kinetic schemes [3] (scheme (4) refers to the potassium channels and (5)—to the sodium ones)

$$n_0 \underset{\beta_n}{\overset{4\alpha_n}{\rightleftarrows}} n_1 \underset{2\beta_n}{\overset{3\alpha_n}{\rightleftarrows}} n_2 \underset{3\beta_n}{\overset{2\alpha_n}{\rightleftarrows}} n_3 \underset{4\beta_n}{\overset{\alpha_n}{\rightleftarrows}} n_4 \tag{4}$$

$$\begin{array}{ccccc}
m_0h_0 & \underset{\beta_m}{\overset{3\alpha_m}{\rightleftarrows}} & m_1h_0 & \underset{2\beta_m}{\overset{2\alpha_m}{\rightleftarrows}} & m_2h_0 & \underset{3\beta_m}{\overset{\alpha_m}{\rightleftarrows}} & m_3h_0 \\
\alpha_h \uparrow \downarrow \beta_h & & \alpha_h \uparrow \downarrow \beta_h & & \alpha_h \uparrow \downarrow \beta_h & & \alpha_h \uparrow \downarrow \beta_h \\
m_0h_1 & \underset{\beta_m}{\overset{3\alpha_m}{\rightleftarrows}} & m_1h_1 & \underset{2\beta_m}{\overset{2\alpha_m}{\rightleftarrows}} & m_2h_1 & \underset{3\beta_m}{\overset{\alpha_m}{\rightleftarrows}} & m_3h_1
\end{array} \tag{5}$$

where α and β are the potential functions [6] and are describing the probability of transfer of one channel between states. The main equation of the model gets a new form [3]:

$$C\frac{dV}{dt} = I - g_{Na}\,[m_3 h_0]\,(V - V_{Na}) + g_K\,[n_4]\,(V - V_K) + g_L(V - V_K) \quad (6)$$

where $[m_3 h_0]$ and $[n_4]$ refer to the number of gates in the open states of sodium and potassium channels, respectively.

2.1 Stochastic version of the model

It has been already shown that using Markov kinetic schemes gives an opportunity of simple transfer of the model from the deterministic to stochastic description [10]. Using description shown in [9] or [10] we can take the number of gates Δn_{AB} that transfer from state A to state B (see scheme (8)) from the binomial distribution (9), where n_A and n_B refer to the number of gates in state A and B, respectively. Probability of transfer of one gate from state A to state B in moment of time between t and $t+\Delta t$ is described with value $p = \Delta t \cdot n_{AB}$. Under some assumptions [4] we can approximate the binomial distribution with the normal distribution $N(\mu, \sigma)$, where $\mu = \Delta n_{AB} \cdot p$ and $\sigma = \Delta n_{AB} \cdot p \cdot (1 - p)$.

$$n_A \underset{r_{BA}}{\overset{r_{AB}}{\rightleftharpoons}} n_B \quad (7)$$

$$P(X = \Delta n_{AB}) = \binom{n_A}{\Delta n_{AB}} p^{\Delta n_{AB}} (1 - p)^{n_A - \Delta n_{AB}} \quad (8)$$

3 Random number generation system overview

Our proposal of random number generation system combines solution based on fast parallel pseudorandom algorithm and true random generator with high entropy. Advantages of one neutralize disadvantages of the other one.

The main idea of our system is a fast parallel pseudorandom generator with periodically replaced seed. New seed is taken from a true random number generator, and swaps old seed of parallel pseudorandom generator just before it hits its period. Our system can work with any generator—in our test implementation we use generator based on inverter rings [13,14] implemented in a FPGA. Parallel pseudorandom algorithm was implemented with the NVIDIA CUDA C programming language and runs on the GPU [2].

3.1 Implementation

The system architecture is shown in Fig. 1. Main part of it is a PC class computer equipped with the NVIDIA GPU with the CUDA technology. Computing capability

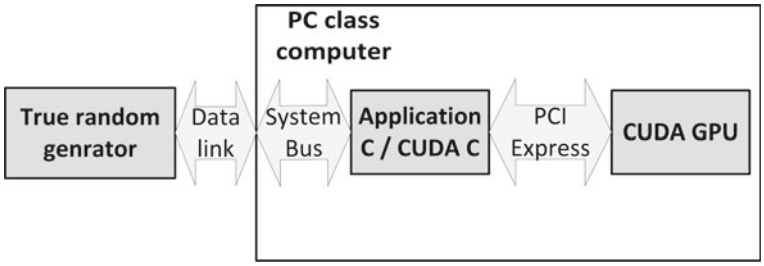


Fig. 1 Random data generation system

of this GPU had to be 2.0 or higher—because we need to use an asynchronous GPU memory access [2] in order to swap seed of a running pseudorandom algorithm.

The true random generator is connected to a PC class computer through the data link. The software application controls work of the parallel GPU algorithm and reads data from the true random number generator. Just before the GPU pseudorandom generator hits its period, the control application swaps its seed. Details about the swapping algorithm are discussed in the following subsection.

Some works show that implementation of a pseudorandom generator in the FPGA can be more efficient [15]. However for our purpose—the simulations of neural cells with use of GPUs—this way of implementation requires transmission of a large amounts of the random data between FPGA and a graphic card through the PCI bus which is very slow [2].

3.1.1 Seed replace algorithm

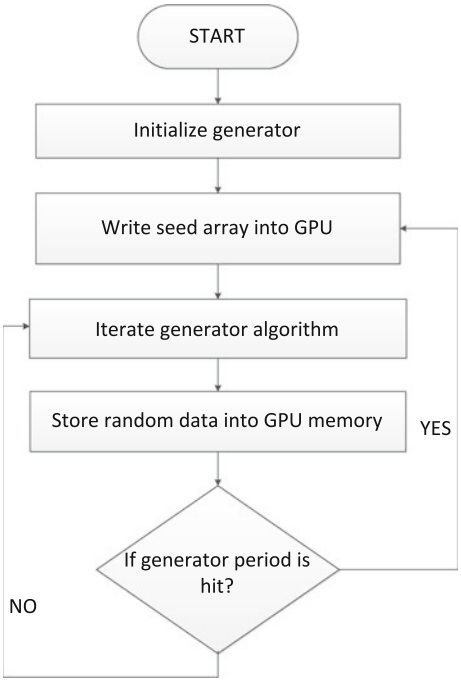
Parallel pseudorandom algorithm with a periodically seed replace mechanism is depicted in Fig. 2. Whole runtime (GPU and CPU part) is controlled by a CPU application. This is a multi-threaded application where one thread is responsible for monitoring GPU part, and the other one, in the mean time, reads random data from the true random number generator and prepares seeds arrays. Every pseudorandom algorithm implemented in GPU works in parallel with many threads. Each thread requires its own seed, thus seeds delivered into GPU are in fact seed arrays.

A generation thread reads the seed at the beginning of each generation loop and it is the only critical section of the algorithm. It must be assured that the seed data stays consistent while it is being read by the generation thread.

4 Approximation of the Gaussian distribution algorithm overview

Second part of the solution presented in this paper focuses on the approximation of Gaussian distribution algorithm. It is based on the set of uniform distribution generators. Output value is calculated as a mean value of draws from the set of uniform generators. Detailed description of mathematics and implementation is depicted in following subsections.

Fig. 2 The seed swap algorithm



4.1 Mathematical background

We are considering a new generator of random numbers from approximation of Gaussian distribution, created with use of N given uniform generators. We will assume that the only known parameters are N—the number of uniform generators and σ—the standard deviation of the Gaussian distribution, as any mean value μ can be obtained simply by shifting samples by wanted value. In Fig. 3 we are showing a simple scheme of arrangement of component uniform generators, where N is the number of generators and x determines the range of the smallest generator.

Each new number x_G from the Gaussian distribution is calculated as a mean value of N numbers taken from each uniform generator:

$$x_G = \frac{x_1 + x_2 + \ldots x_N}{N}, \tag{9}$$

where x_k is a number taken from kth generator, for $k = 1, 2, \ldots, N$. For each component uniform generator we can count the mean value and variance of the distribution:

$$\mu_k = EX_k = \sum_{i=0}^{kx} i \cdot \frac{1}{kx+1} = \frac{kx}{2}, \tag{10}$$

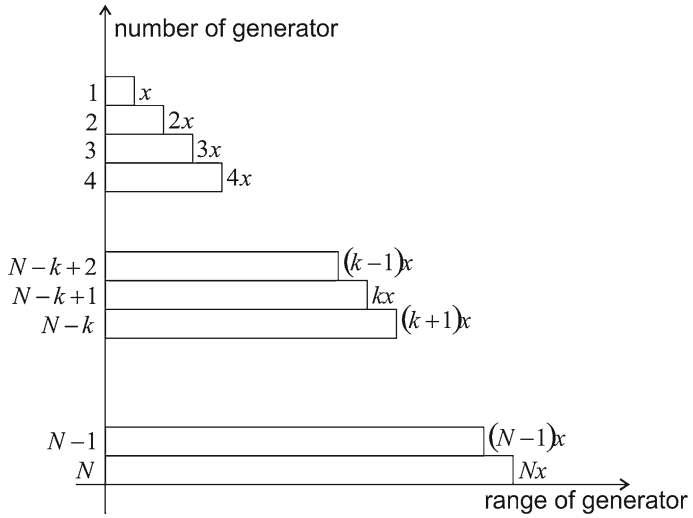


Fig. 3 Scheme of N uniform generators

$$\begin{aligned}\sigma_K^2 &= D^2 X_k = EX_k^2 - (EX_k)^2 \\ &= \sum_{i=0}^{kx} i^2 \cdot \frac{1}{kx+1} - \left(\sum_{i=0}^{kx} i \cdot \frac{1}{kx+1}\right)^2 \\ &= \frac{kx\,(kx+1)}{12}. \end{aligned} \tag{11}$$

With counted μ_k and σ_k^2 of a single uniform distribution, it is now easy to obtain the mean value μ and standard deviation σ of our Gaussian generator:

$$\mu = \sum_{k=1}^{N} \mu_k \cdot \frac{1}{N} = \sum_{k=1}^{N} \frac{kx}{2} \cdot \frac{1}{N} = \frac{x\,(N+1)}{4}, \tag{12}$$

$$\begin{aligned}\sigma &= \frac{\sqrt{\sigma_1^2 + \sigma_2^2 + \ldots + \sigma_N^2}}{N} = \frac{\sqrt{\sum_{k=1}^{N} \sigma_k^2}}{N} \\ &= \frac{\sqrt{\sum_{k=1}^{N} \left(\frac{kx(kx+1)}{12}\right)^2}}{N}. \end{aligned} \tag{13}$$

Above Eq. (12) and (13) allow us to form the following set of equations for μ and x:

$$\begin{cases} -2\,(2N+1)\,\mu^2 - 3\,(N+1)\,\mu + 9N\sigma^2\,(N+1) = 0 \\ x = \dfrac{4\mu}{N+1} \end{cases} \tag{14}$$

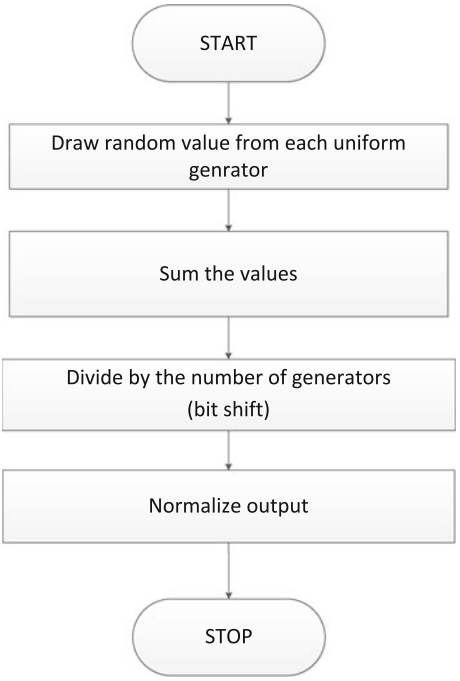


Fig. 4 The approximation algorithm

where x is given explicitly, while for μ it is necessary to solve the quadratic equation.

4.2 Implementation

The algorithm was implemented on GPU using CUDA technology. It consists of a set of parallel pseudorandom generators with the swapped seeds. These generators produce uniform distributed stream of random data. In order to achieve approximated Gaussian distributed value, algorithm calculates the mean value from values draw from each uniform generator in the set. Furthermore, to simplify calculations, number of uniform generators N in the set, and variance σ of the output is always selected as the power of two, thus divisions in the mean value calculation and normalization are implemented as, simple and fast, bit shifts. The approximation algorithm is depicted in Fig. 4.

5 Tests of the sample implementations

This section presents tests of the sample implementations of the system. Next subsection shows the results of a statistical tests of the generation system with a periodically swapped seed, while the second one focuses on the quality tests of the approximation algorithm. The last subsection presents usage of the sample implementation of the complete generating system (consisting of a both mentioned parts) in a neural cells simulation task.

5.1 Random generator system

Quality of a random data generated with our system was rated with statistical tests from packets: DIEHARD [7] and SP800-22 [8]. For the test purpose we choose a parallel GPU implementation of the Xorshift with swapped seeds taken from the true random number generator based on inverter rings [13,14] implemented in Xilinx Virtex 5 FPGA. The tests were run five times, and for every run a new set of random data was generated. The results depicted in the next part of this subsection are mean results of all five runs of a test.

Figure 5 shows results of a tests from the DIEHARD packet. In this case random sequences of 3,000,000 of a 32 bits values were tested. On the graph two statistical significance levels were marked with the use of four lines. Red lines indicate statistical significance level $\alpha = 0.01$, while green indicate level $\alpha = 0.05$. Blue dots mark p-values calculated in particular test. In some test cases, it is possible to calculate final p value. It is done with Kolmogorov–Smirnov test and is marked with red dot. If the most of the points is placed between lines signalizing actual significance level the test is considered as passed, which is shown on the graph with the bold caption. In this particular case examined random number generator has passed all tests.

Tests from SP800-22 packet were run on 3,122,000 of 32 bits random values. The statistical significance level was set to $\alpha = 0.01$, and is marked on the graph with red lines. Test parameters were set according to documentation [8]. Results are shown in Fig. 6, and should be interpreted as in the DIEHARD case.

5.2 Approximation algorithm

5.2.1 χ^2 test of goodness

This part of the paper we have devoted for the considerations on χ^2 test of goodness, as the most popular and commonly used test. This test can be any statistical test, which statistics has the χ^2 distribution, when founded null hypothesis is true. The χ^2 test, by putting the null hypothesis, gives the possibility to verify the normality of tested data.

In Table 1 we gained the probabilities with which it is possible to reject the founded null hypothesis. If so, probability close to value 0 gives a good chance to accept the null hypothesis and at the same time—rejects that considered generator gives values from Gaussian distribution. Similar, values of probability close to value 1 confirms that we are dealing with the normal distribution.

Probabilities gained in Table 1 show how the results varied from each other. We have tested the built in MATLAB environment generator `normrnd` and commonly used Box-Muller generator [1], with both—sine and cosine—options, in comparison with our generator, for different number of uniform component generators ($N = 2$, 350, 2 000). For each of tested generators, we performed 10 independent attempts, each time we counted the mean value of gained data. The results show that, for small number N of uniform generators the mean value of probability is very small, which

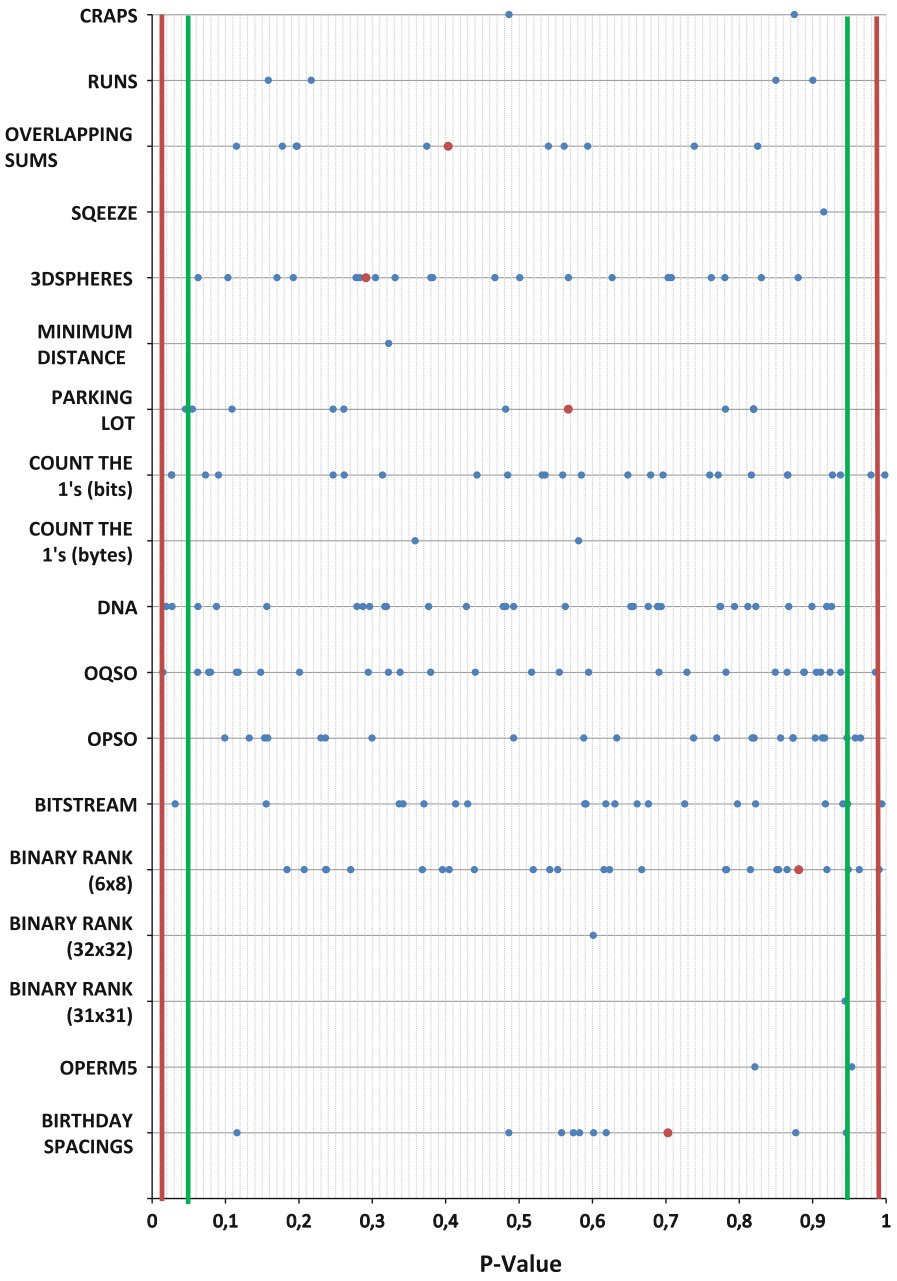


Fig. 5 DIEHARD test results for Xorshift generator with periodically swapped seed

makes us to reject our assumption of the normality of our data. It is also easy to notice, that for higher number of generators the mean value of probability is growing, and for $N = 2{,}000$ this mean value is greater than for commonly used Box-Muller generator,

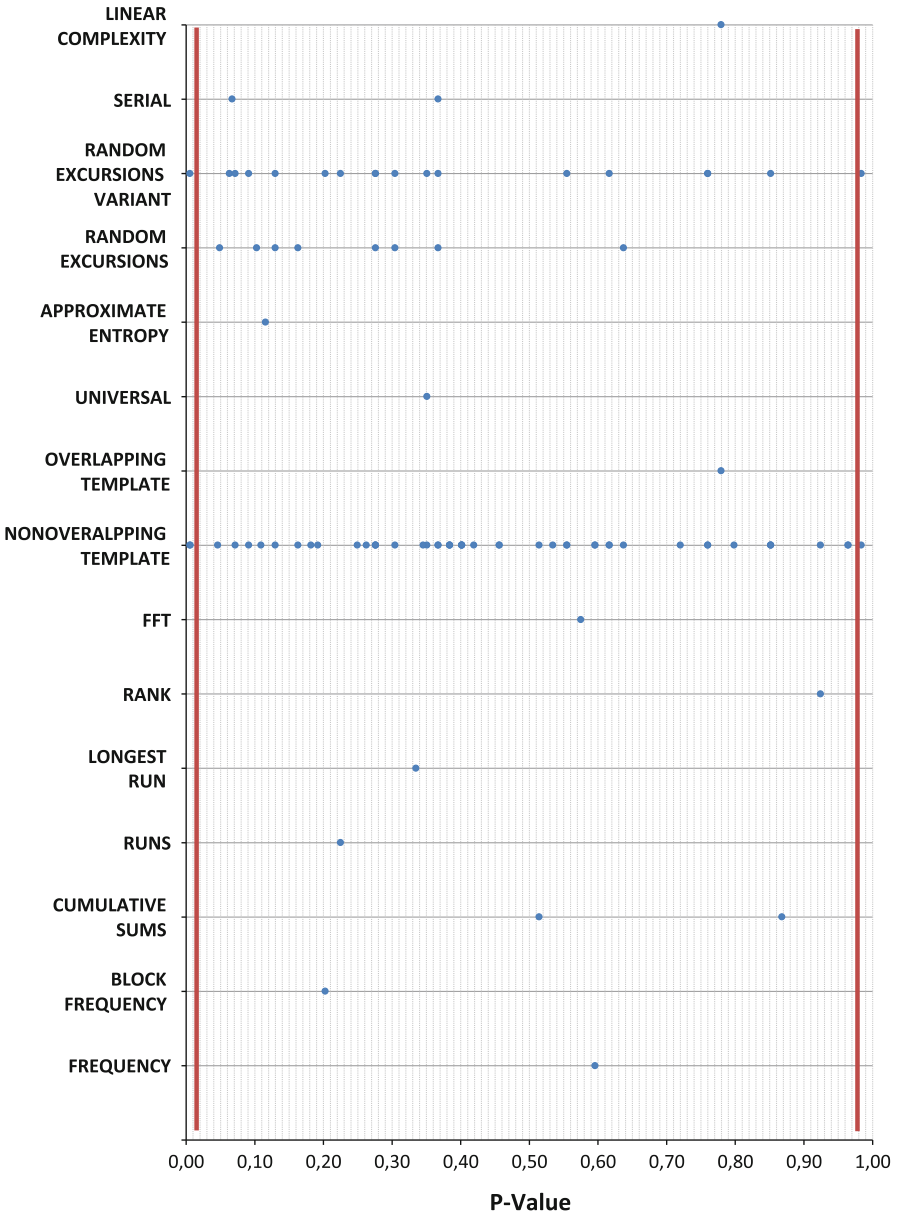


Fig. 6 SP800-22 test results for Xorshift generator with periodically swapped seed

what gives the assumption that for even higher number of uniform generators we will be able to obtain the probability value close to 1.

In the next section, we will also show histograms for chosen numbers of generators, where it will be possible to compare data from our generator with the real Gaussian distribution curve.

Table 1 Probabilities

No. of attempt	Matlab normrnd	Box-Muller sinus	Box-Muller cosinus	Our generator		
				$N = 2$	$N = 350$	$N = 2{,}000$
1	0.0674	0.7297	0.5024	0	0.3867	0.0387
2	0.7060	0.3838	0.2947	0	0.1958	0.9801
3	0.0688	0.4028	0.3683	0	0.0005	0.8320
4	0.7121	0.7687	0.7022	0	0.4862	0.1109
5	0.4274	0.6356	0.2888	0	0.0020	0.9337
6	0.7767	0.6701	0.3087	0	0.1869	0.9285
7	0.5033	0.3105	0.9662	0	0.0028	0.7321
8	0.7854	0.1348	0.0350	0	0.0021	0.4289
9	0.9566	0.0501	0.4856	0	0.2353	0.6371
10	0.7536	0.8635	0.2280	0	0.0041	0.9003
Mean value	0.5757	0.4950	0.4180	0	0.1502	0.6522

5.2.2 Histograms

One of the ways of analysing of gained data are the histograms. Before the histogram is being drawn it is necessary to pick out the number of bins. There is a lot of different rules of choosing the number of bins, depending on features that we want to extract from the data. We used the Struges' rule [11,12]:

$$k = \lceil \log_2 n + 1 \rceil \tag{15}$$

where n refers to the number of generated probes. The Struges' rule is one of the most common rules, used to analyze the normality of data. In our case $n = 10{,}000{,}000$, thus $k = 25$.

In Fig. 7 there are shown four histograms created from data taken from our generator, with the assumption of different number of component uniform generators ($N = 2,\ 3,\ 350,\ 2{,}000$). Additionally, for comparison, in Table 2 there are shown the differences between histograms and the Gaussian curve for different analyzed generators (MATLAB `normrnd`, Box-Muller).

Data collected from our generator had to be normalized. It was performed by dividing the values of bars by the area beneath the curve designated by the histogram. New values of bars are used to determine the error between the histogram and the Gaussian curve. This error is counted as a sum of differences between the value of each bar and the corresponding value of Gaussian distribution. Either the histograms (Fig. 7) or the errors (Table 2), confirms that with properly picked number of component uniform generators, our generator does not deviate from results obtained with MATLAB `normrnd` or Box-Muller generators.

The huge advantage of our generator is the possibility of parallelization of calculations and thus speed up of generation of huge number of data.

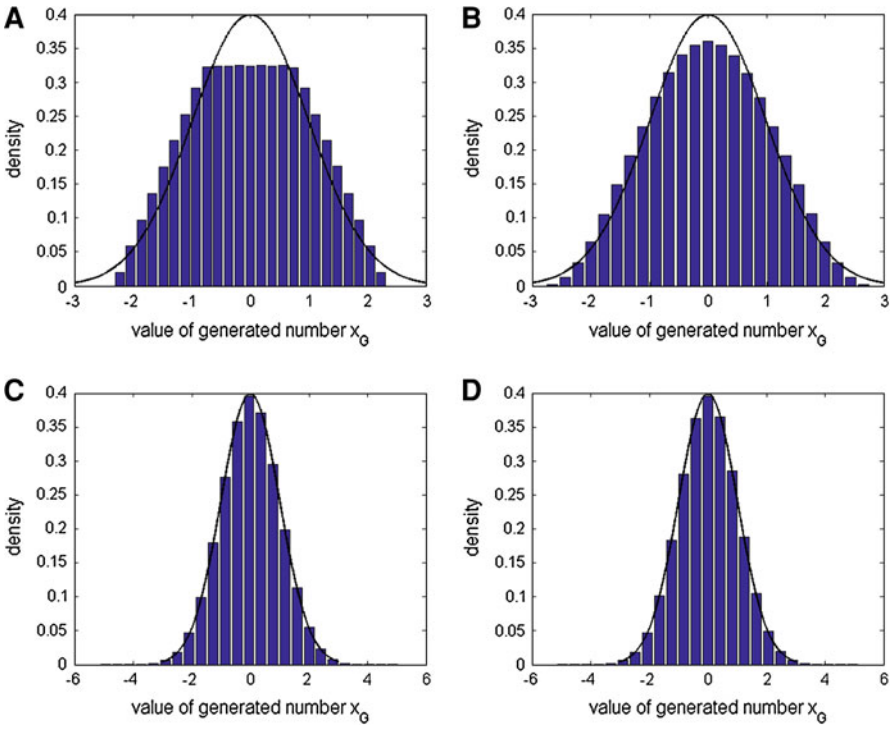


Fig. 7 A. $N = 2$ B. $N = 3$ C. $N = 350$ D. $N = 2{,}000$

Table 2 Probabilities

No. of attempt	Matlab normrnd	Box-Muller sinus	Box-Muller cosinus	Our generator		
				$N = 3$	$N = 350$	$N = 2{,}000$
1	0.0155	0.0172	0.0169	0.4332	0.0179	0.0183
2	0.0165	0.0162	0.0175	0.4328	0.0174	0.0179
3	0.0155	0.0168	0.0171	0.4339	0.0190	0.0173
4	0.0165	0.0163	0.0175	0.4345	0.0173	0.0174
5	0.0170	0.0165	0.0170	0.4327	0.0184	0.0183
6	0.0163	0.0172	0.0174	0.4332	0.0177	0.0176
7	0.0166	0.0165	0.0167	0.4332	0.0190	0.0183
8	0.0160	0.0175	0.0163	0.4315	0.0193	0.0181
9	0.0188	0.0198	0.0176	0.4340	0.0185	0.0180
10	0.0166	0.0186	0.0180	0.4326	0.0177	0.0180
Mean value	0.0165	0.0173	0.0172	0.4332	0.0182	0.0179

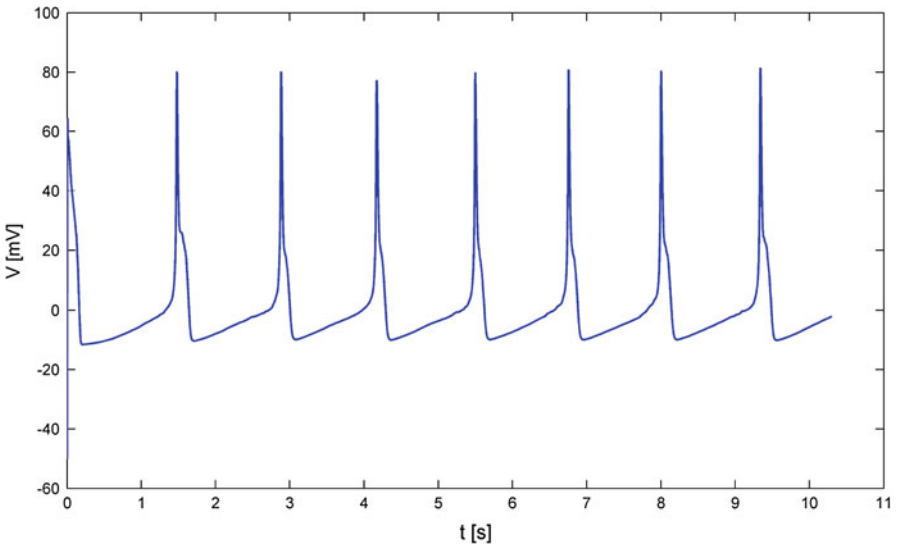


Fig. 8 Model output of tested neural cell model

5.3 Combined system in a neural cell simulation task

Last part of the tests was, in our point of view, the most important. It relies on use of our system in neural cell simulation. The model of the cell was based on Hodgkin–Huxley stochastic model. We used combined system of the parallel pseudorandom generator with swapped seed and approximation of Gaussian distribution algorithm. We evaluate shape of action potential curve of a simulated cell. Input current and internal cell parameters of all tests were identical and do not influence on the results. Test model was build with Xorshift parallel random generator, true random generator based on inverter rings implemented in FPGA and approximation of Gaussian distribution algorithm with varying number of uniform generators.

Figure 8 presents model output curve and Figs. 9, 10 and 11 show action potential obtained from test cell model with various number of uniform generators—respectively 2, 4, and 8. Model output was obtained from the stochastic cell model based on true number generator, where the Gaussian distribution was obtained from the Box-Muller transform.

6 Summary and conclusion

Test show that simple and fast random number generator properties can be improved by use of the system. In our example parallel implementation of Xorshift pseudorandom generator pass very restrictive statistical tests. This solution combines profits of high entropy hardware random generators with speed of parallel pseudorandom algorithms.

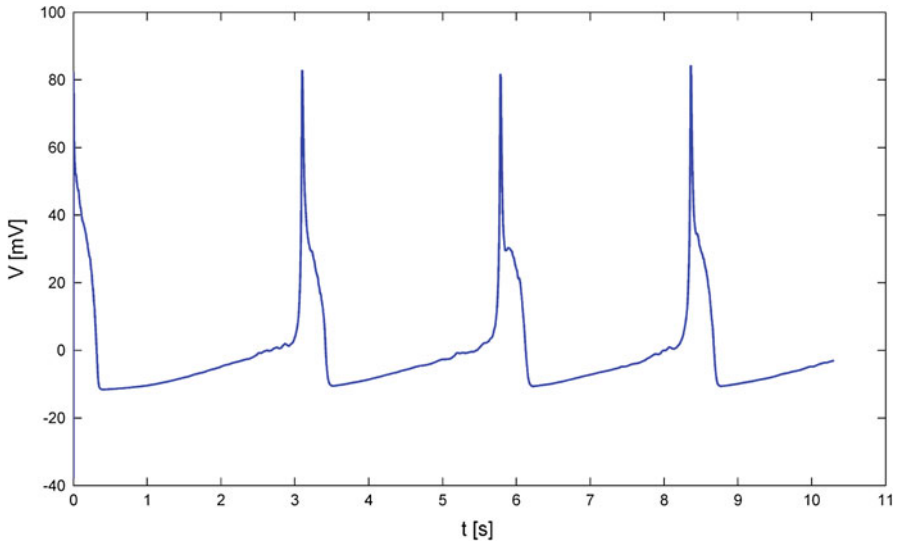


Fig. 9 Output of tested neural cell model with two uniform generators

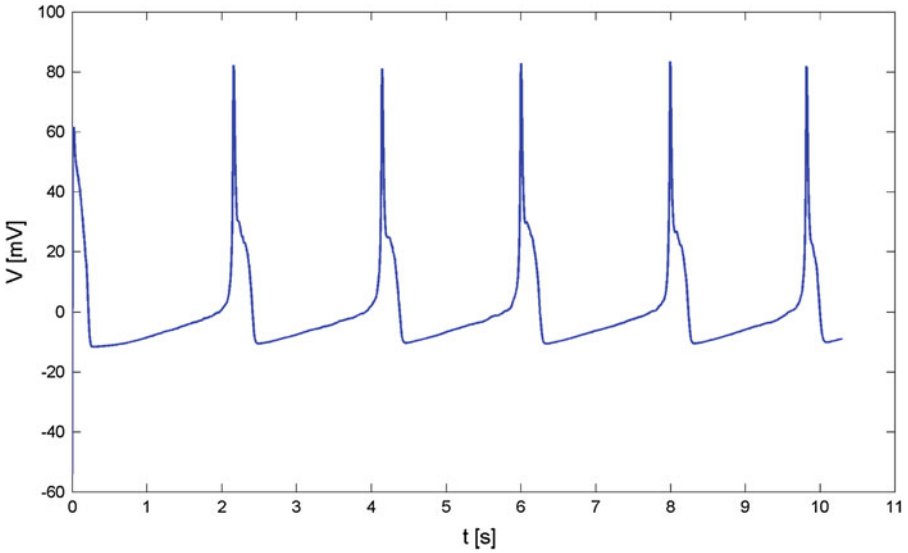


Fig. 10 Output of tested neural cell model with four uniform generators

Approximation of Gaussian distribution algorithm is fast alternative to commonly used Box–Muller transformation. Our test showed that quality of generated random stream is accurate and could be used in various areas.

The last test proved usability of proposed system in neural cell simulations based on stochastic version of Hodgkin–Huxley model. Using only eight uniform distribution generators we were able to produce output signal as good as model.

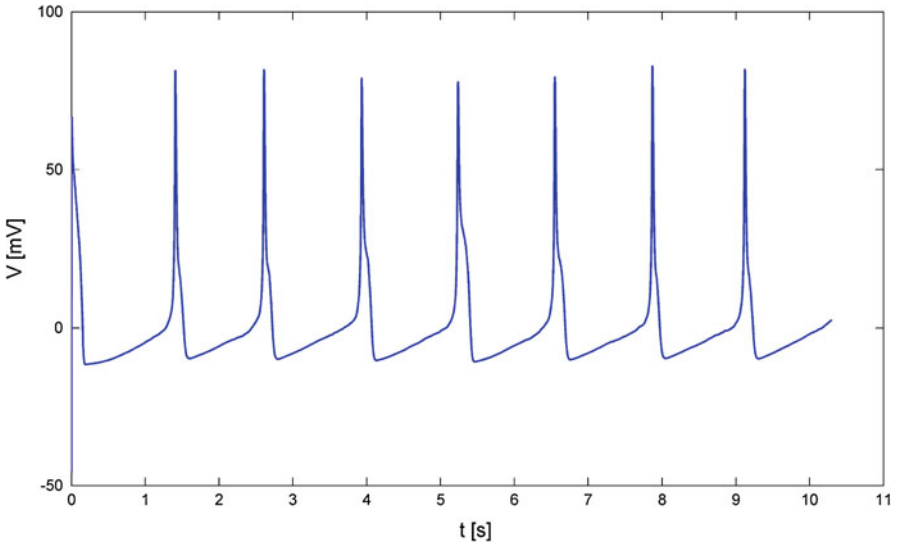


Fig. 11 Output of tested neural cell model with eight uniform generators

Concluding, proposed system proves its usability and functionality in neural cells simulation tasks. Using this solution could give benefits of faster simulations without loosing accuracy.

References

1. Box GEP, Muller ME (1958) A note on the generation of random normal deviates. Ann Math Stat 29(2):610–611
2. NVidia corp (2012) NVIDIA CUDA C Programing Guide version 4.0. NVidia.
3. Destexhe A, Mainen ZF, Sejnowski TJ (1994) Synthesis of models for excitable membranes, synaptic transmission and neuromodulation using a common kinetic formalism. J Comput Neurosci 3(1): 195–230
4. William F (1968) An introduction to probability theory and its applications vol 1, chapter VII. Wiley, New York
5. Gugała K, Swietlicka A, Jurkowlaniec A, Rybarczyk A (2011) Parallel simulation of stochastic denritic neurons using nvidia gpus with cuda c. Elektronika konstrukcje technologie zastosowania 12(2011): 59–61
6. Hodgkin AL, Huxley AF (1952) A quantitative description of membrane current and its application to conduction and excitation in nerve. J Physiol 117:500–544
7. Marsaglia G (1995) The marsaglia random number cdrom (online).
8. Rukhin A, Soto J, Nechvatal J, Smid M, Barker E, Leigh S, Levenson M, Banks D, Vangel M, Heckert A, Dray J, Vo S (2010) A statistical test suite for random and pseudorandom number generators for cryptographic applications. 800–22 (Revision 1a).
9. Saarinen A, Linne M-L (2006) Modeling single neuron behavior using stochastic differential equations. Neurocomputing 69:1091–1096

10. Schneidman E, Freedman B, Segev I (1998) Ion channel stochasticity may be critical in determining the reliability and precision of spike timing. Neural Comput 10(7):1679–1703
11. Scott DW (2009) Struges' rule. Wiley interdisciplinary Reviews: computational statistics, vol 1. pp 303–306
12. Struges HA (1962) The choice of a class interval. J. Am Stat Assoc 65–66.
13. Stinson DR, Sunar B, Martin WJ (2007) A provably secure true random generator with build-in tolerance to active attacks. IEEE Transact 56(1):109–119
14. Tan CH, Wold K (2009) Analysis and enchancement of random number generator in FPGA based on oscillator rings. Int J Reconfig Comput (article ID 501672).
15. Tian X, Benkrid K (2009) Mersenne twister random number generation on FPGA, CPU and GPU. Adaptive Hardware and Systems, 2009. AHS 2009. NASA/ESA Conference on, pp 460–464.