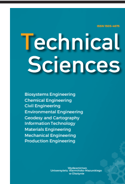
Technical Sciences, 2021, 24, 331–349

Yearbook peer-reviewed scientific journal
ISSN 1505-4675
e-ISSN 2083-4527
TECHNICAL SCIENCES
Homepage: https://czasopisma.uwm.edu.pl/index.php/ts/



DOI: https://doi.org/10.31648/ts.7238

PSEUDO-RANDOM NUMBER GENERATOR BASED ON LINEAR CONGRUENCE AND DELAYED FIBONACCI METHODE

Radosław Cybulski

ORCID: 0000-0003-1289-5318
Chair of Mathematical Method in Computer Science
University of Warmia and Mazury in Olsztyn

Received 11 November 2021, accepted 04 December 2021, available online 06 December 2021.

Key words: Linear congruential method, Delayed Fibonacci technique, Hybrid pseudo-random number generator.

Abstract

Pseudo-random number generation techniques are an essential tool to correctly test machine learning processes. The methodologies are many, but also the possibilities to combine them in a new way are plenty. Thus, there is a chance to create mechanisms potentially useful in new and better generators. In this paper, we present a new pseudo-random number generator based on a hybrid of two existing generators – a linear congruential method and a delayed Fibonacci technique. We demonstrate the implementation of the generator by checking its correctness and properties using chi-square, Kolmogorov and we apply the Monte Carlo Cross Validation method in classification context to test the performance of the generator in practice.

Introduction

First question is what is it pseudo-random generator. It is generator which create a numbers in randomness method. To do that we need pseudo-random generator with good seed. But now is the question what is seed. The seed is the starting value on the basis of which the remaining numbers are generated.

Correspondence: Radosław Cybulski, Katedra Metod Matematycznych Informatyki, Wydział Matematyki i Informatyki, ul. Słoneczna 54, 10-710 Olsztyn, e-mail: radoslaw.cybulski@uwm.edu.pl

Seed must have features two unpredictability, first in forward saying that other who's don't know seed can't create a good pseudo-random numbers. Second feature relies on that other can't create seed when knows created numbers. This features says why seed is that important in pseudo-random generators. When we looking on seeds in generators existing to create our method we can see different between, but it will be show in this introduction. Second question is how we can use pseudo-random generators. This generator can be used to Monte Carlo methods simulation, cryptography, and to create computer games when we need use randomness.

We will briefly introduce the algorithms on the basis of which we create our hybrid pseudo-random number generation method. Then, we will proceed with an overview of selected existing generators.

1. Linear congruence generator (STALLINGS 2012)

The generation of pseudo-random numbers using a linear congruence generator is based on an iterative algorithm

$$X_{n+1} = (aX_n + c) \bmod m,$$

where individual parameters have the following meaning:

$$m \text{ – module, } m > 0,$$

$$a \text{ – multiplier, } 0 < a < m,$$

$$c \text{ – growth } 0 <= c < m,$$

$$X0 \text{ – starting value (seed) } 0 <= X0 < m.$$

Each pseudo-random number being generated is from 0 to $m - 1$ interval. It is worth noting here that only integers are used for the generator. For the quality of the generator functioning in this way it is critical to choose the values of a, c and m. The main advantage of the linear congruence generator is the choice of two parameters, which are the multiplier (a) and the modulus (m), where it can happen that the generated sequence of numbers will be indistinguishable from the sequence of numbers that arise because of random drawing without returning numbers from the set $\{1, 2, ..., m - 1\}$.

The disadvantage is that if the intruder knows our parameters, by using which the generation of pseudo-random numbers occurs, he can map the generated sequence himself. He will do it using the following equations:

$$X_1 = (aX_0 + c) \bmod m,$$

$$X_2 = (aX_1 + c) \bmod m,$$

$$X_3 = (aX_2 + c) \bmod m.$$

2. Delayed Fibonacci generator (Lagged Fibonacci Generator, online)

A pseudo-random number generator used to improve the linear congruence generator. Its origins date back to 1958, with contributions from GJ Mitchell and DP Moore. The algorithm is based on the following formula:

$$S_n = S_n - j * S_n - k \ (\mathrm{mod}\ m),\ 0 < j < k,$$

where the operator * can be any arithmetic operation or a bit operation. Depending on the operation used, the generator takes the appropriate name.

The operation of the generator starts with the selection of the j and k number values (they are the selected indices from the seed that take part in the creation of pseudo-random numbers). Then, the value of the number m is determined. The last parameter of the generator is the value val, which is considered as the seed. With this combination, our values will be drawn from the range of 0 to $m - 1$. It is also important to use the m value, which is a power of 2. The generator is delayed, because it remembers several values generated in the previous steps.

Let us discuss the content of the following sections. In the section 2 we have an overview of selected pseudo-random number generation methods. In the section 3 we present the idea of our algorithm. Then in section 4 we test our method using Chi-square and Kolmogorov's lambda consistency test. In the section 4.3 we conduct a critical discussion of our results. In section 5 we present the application of our method in practice. Finally, in section 6 we summarise the work.

We proceed to review what we consider to be the more important pseudo--random number generation techniques.

Selected pseudo-random number generation techniques

In this section we will present a few pseudo-random generator with normal distribution.

1. Mersenne Twister (SULEWSKI 2019)

A popular pseudo-random number generator by Japanese scientists Matsumoto and Nishimura, generating pseudo-random numbers with a uniform distribution of massive period $2^{19937} - 1$. This algorithm provides a solution for generating pseudo-random numbers for many software systems. The generator was created to improve the quality of older generators.

2. Box-Muller (BM) generator (SULEWSKI 2019, BOX, MULLER 1958)

Let U_1, U_2 be $U(0, 1)$ pseudo-random numbers. The Box-Muller method generates a pair of independent pseudo-random numbers (Y_1, Y_2) from $N(m, s)$:

- $a = \sqrt{-2\ln(U_1)}$, $b = 2\pi U_2$;
- $X_1 = a\sin(b)$, $X_2 = a\cos(b)$;
- $Y_1 = sX_1 + m$, $Y_2 = sX_2 + m$.

3. Polar Generator (Sulewski 2019, Bell 1968, Knop 1969)

Let U_1, U_2 be $U(0, 1)$ pseudo-random numbers. A polar method generates a pair of independent pseudo-random numbers (Y_1, Y_2) from $N(m, s)$:

- $a = -1 + 2 * U_1, b = -1 + 2 * U_2$;
- $d = a^2 + b^2$;
- *If* $d \geq 1$, go to point 1;
- $e = \sqrt{\frac{-2\ln(d)}{d}}$;
- $X_1 = ae, X_2 = be$;
- $Y_1 = s * X_1 + m, Y_2 = s * X_2 + m$.

4. Quotient method (Sulewski 2019, Kinderman, Monahan 1977, Wieczorkowski, Zielinski 1997)

Let U_1, U_2 be the pseudo-random numbers $U(0, 1)$. A quotient method generating a pseudo-random number Y from $N(m, s)$:

- $u = U_1, e = \exp(1), v = -\sqrt{2/e} + 2\sqrt{2/e}U_2$;
- $X = v/u$;
- *If* $X^2 \leq 2(3 - u(4 + u))$, go to point 6;
- *If* $X^2 \leq 2/u - 2u$ and $X^2 \leq -4\ln(u)$, go to point 6;
- Go to point 1;
- $Y = sX + m$.

5. Ahrens-Dieter (AD) generator (Sulewski 2019, Ahrens, Dieter (1988)

Let U_1, U_2, U_3 be $U(0, 1)$ pseudo-random numbers. The Ahrens-Dieter method generates a pair of independent pseudo-random numbers (Y_1, Y_2) from $N(m, s)$:

- *If* $U_1 < 0.5$ then $a = 1$. Go to point 3;
- $a = -1$;
- $b = -\ln(U_2)$;
- $c = \mathrm{tg}[\pi(U_3 - 0.5)]$;
- $d = \sqrt{\frac{2b}{1 + c^2}}$;
- $X_1 = ad, X_2 = cd$;
- $Y_1 = s * X_1 + m, Y_2 = s * X_2 + m$.

Now we turn to discuss the methodological details of our new pseudo-random number generation technique.

Proposed Methodology

First, what we do, to create pseudo-random numbers, is the creation of generator seed. Seed is a message, whose input chars are changed into array ASCII codes. When we have seed, the next step to create pseudo-random numbers is to choose two numbers j and k (both numbers be in range between 0 and length seed, for example if our input message has 10 chars, j and k be in range between 0 and 9). These numbers are indexes from the array ASCII codes. Next we select number m that sets the generator range, if our m is 342, the biggest pseudo--random number will be equal 341. The last thing what we should do creates number n, that will determine how many sequences the generator will perform.

One sequence includes several operations:

Step 1

$$a = (S_j\ XOR\ S_k) \bmod m,$$

$$b = (S_k * S_k) \bmod m;$$

Step 2

$$j = a \bmod S,$$

$$k = b \bmod S;$$

Step 3

$$S_j = a,$$

$$S_k = b.$$

Two pseudo-random numbers (a and b) are created in one sequence. Next when we have created the numbers a and b, the numbers j and k are change based on actions described above. When we have new indexes j and k, old values on new indexes j and k are replaced by two pseudo-random numbers a and b.

Example of generator action

Input message: gerwok-*nkl*

Array ASCII codes from input message (S): 103 101 114 119 111 107 45 110 107 108

$j = 6$

$k = 8$

$m = 16452$

$n = 312$

First sequence

$a = (45\ \text{XOR}\ 107) \bmod 16452 = 70 \bmod 16452 = 70$

$b = (107 * 107) \bmod 16452 = 11449 \bmod 16452 = 11449$

$j = 70 \bmod 10 = 0$

$k = 11449 \bmod 10 = 9$

$S_j = 70$

$S_k = 11449$

$S = 70\ 101\ 114\ 119\ 111\ 107\ 45\ 110\ 107\ 11449$

Second sequence
a = (70 XOR 11449) mod 16452 = 11519 mod 16452 = 11519
b = (11449 ∗ 11449) mod 16452 = 131079601 mod 16452 = 6517
j = 11519 mod 10 = 9
k = 6517 mod 10 = 7
S_j = 11519
S_k = 6517
S = 70 101 114 119 111 107 45 6517 107 11519
That will be created 624 pseudo-random numbers.

1. Illustration of examples of generated pseudo-random numbers

In this section, we shall present graphs of the distribution of pseudo-random numbers, A and B. The numbers generated using the three input sets will be presented. For each set two graphs will be presented, where one will show on the x-axis the number a and on the y-axis the number b. The second graph will be the inverse of the first one on the x-axis the number b and on the y-axis the number a. Additionally, we will show graphs presenting the numbers of generated pseudo-random numbers in intervals. The number of intervals, the size of the interval and the first upper limit of the interval is determined by:

$$k \approx \sqrt{n} + 1,$$

where:
k – number of intervals,
n – the number of pseudo-random numbers generated;

$$h \approx \frac{r}{k},$$

where:
h – interval size,
r – the difference between the largest and smallest pseudo-random numbers generated,
k – number of intervals;

$$P_1 \approx n_{\text{min.}} - \left(\frac{h}{2}\right) + h,$$

where:
P_1 – right limit of the first interval,
$n_{\text{min.}}$ – the smallest pseudo-random number generated,
h – size of the interval.

2. Example 1

Input message: gerwok-*nkl*
j = 6
k = 8
m = 16452
n = 84

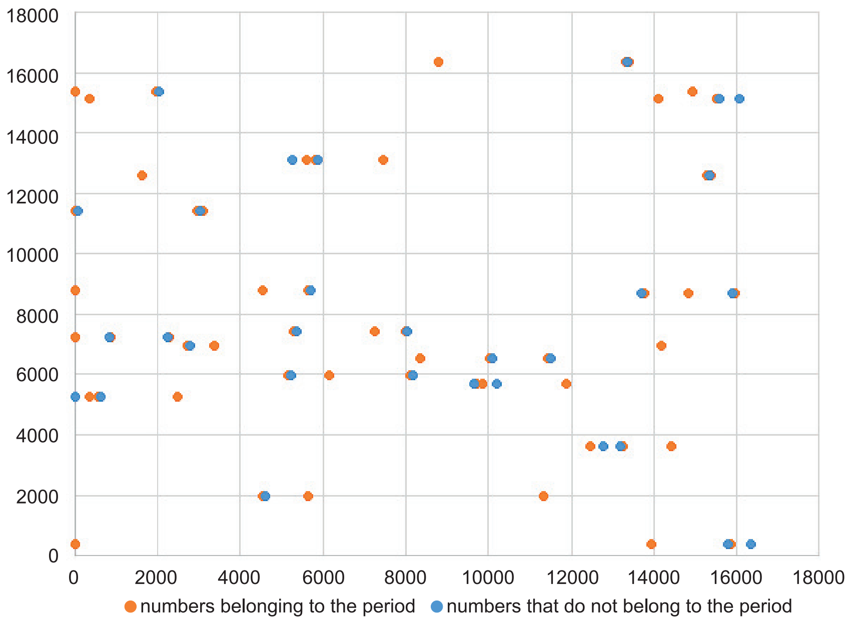


Fig. 1. Plot of distribution pseudo-random numbers A on the x-axis and pseudo-random numbers B on the y-axis

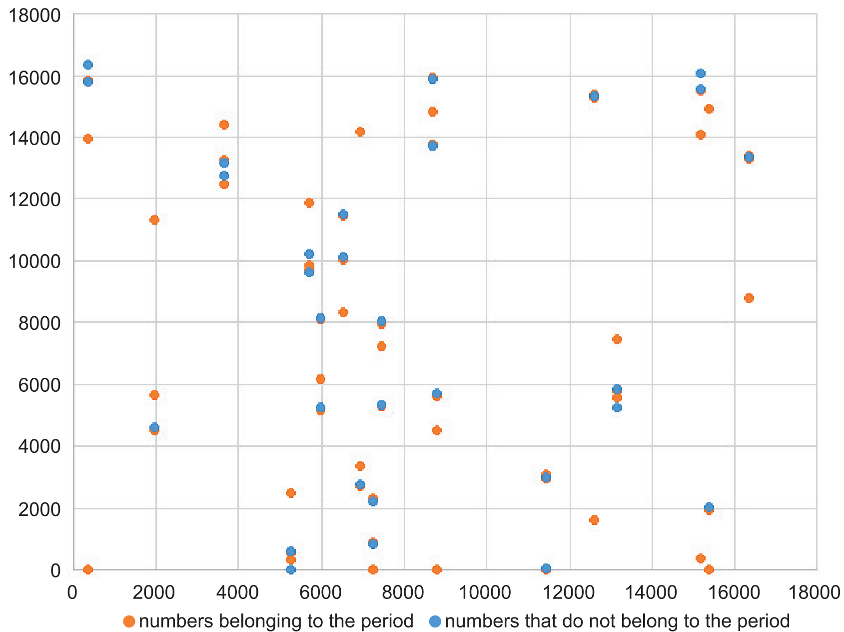


Fig. 2. Plot of distribution pseudo-random numbers B on the x-axis and pseudo-random numbers A on the y-axis

Looking at the graphs (Figs. 1, 2) it can be seen that not all points that were created during the generation of pseudo-random numbers were placed on the graph. The reason is that a generator period generates a repeating sequence of pseudo-random numbers from a certain point onward. In this example, the period starts with the 31st generator pass, up to this point points not belonging to the repeating sequence have been generated. While the length of the generator period for this example is 54. The distribution of the points itself shows a non--uniform distribution. If we analyze the generate pseudo-random numbers *A* and *B*, we can see that for the pseudo-random numbers *B* a repeating sequence of numbers has been formed, the length of which is 18.

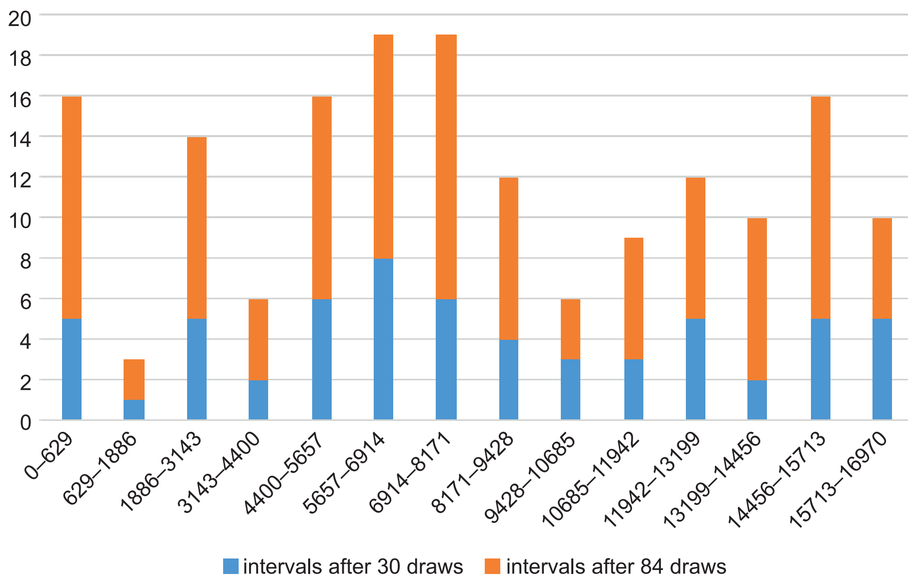


Fig. 3. Interval frequency histogram from generated numbers

The graph (Fig. 3) showing the distribution in numerical intervals shows the frequencies of pseudo-random numbers in each interval divided into two series. The first blue shows the distribution of numbers after 30 draws, and the second orange after 84 draws. The whole graph shows an uneven distribution of the intervals. Only after 30 draws we can observe small differences between the intervals. Although the generated unique points were used to build the counts (i.e. not repeating both values of *A* and *B* during the 84 draws, only after the next pass we will get the numbers *A* and *B* in the same configuration: the point created in the generation number 32 will be repeated in the generation number 85), we can see a large variation, which can be caused by the repeated values of the number *B* from time to time.

3. Example 2

Input message: &*%06frD34>”;/[]234#^(&

$j = 6$

$k = 20$

$m = 391$

$n = 138$

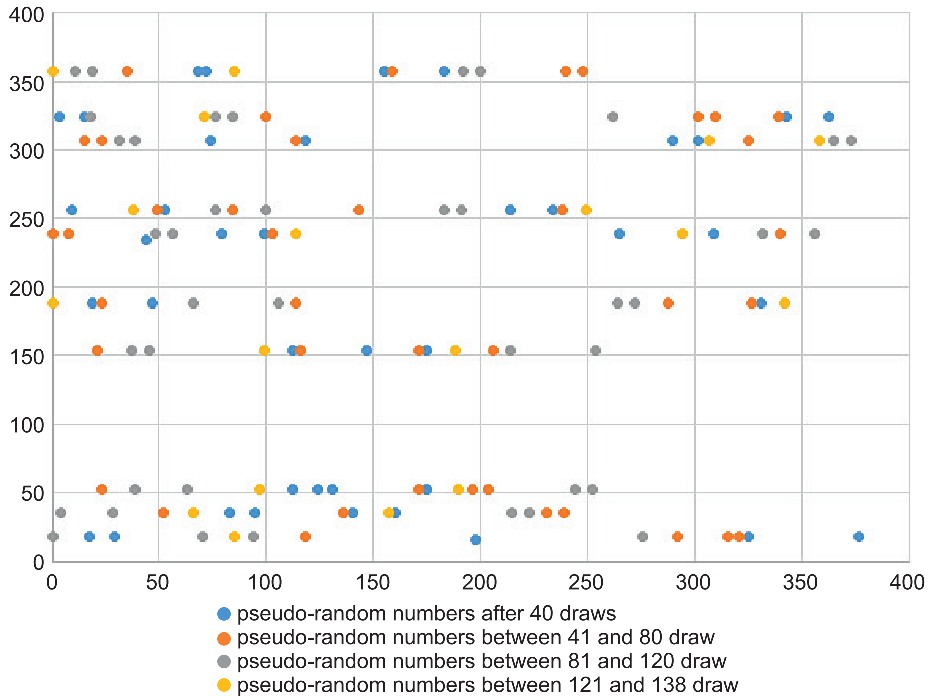


Fig. 4. Plot of distribution pseudo-random numbers A on the x-axis and pseudo-random numbers B on the y-axis

The graphs (Figs. 4, 5) shown for the second example show a more true random distribution. The points marked with four colors show the distribution of pseudo-random numbers after a certain number of generations. For the second example, we also notice some distribution of points in intervals (the pseudo--random number B falls into a generator period of length 10), but the points themselves are unique, because after 138 steps of the generator, the first repetition of a point occurring will be recorded (20 points will be repeated). The generator period itself will start at 120 draws, thus if the parameter n is greater than 138 for the dataset, points from 120 draws will start to repeat.

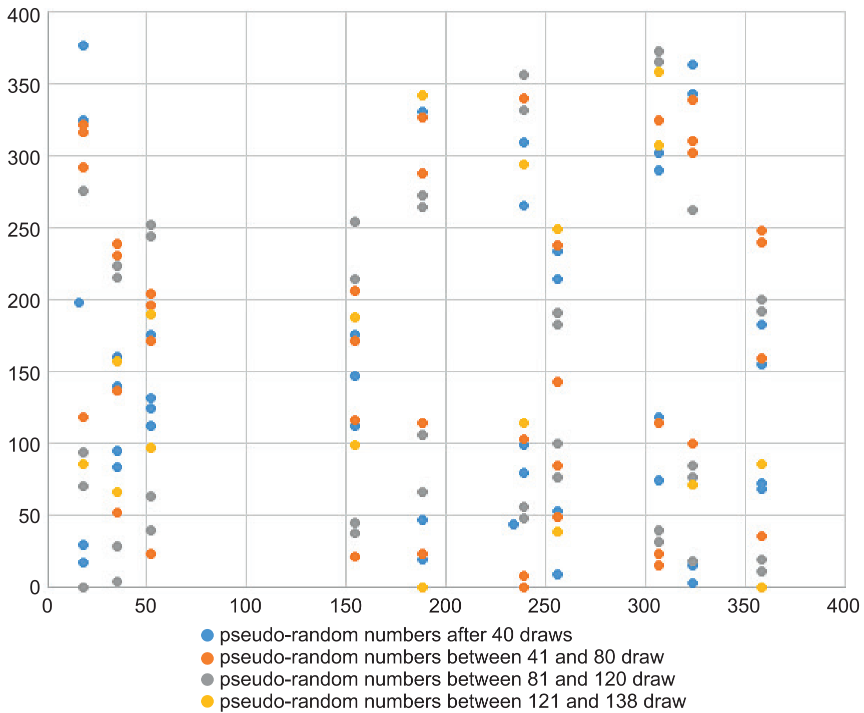


Fig. 5. Plot of distribution pseudo-random numbers B on the x-axis and pseudo-random numbers A on the y-axis

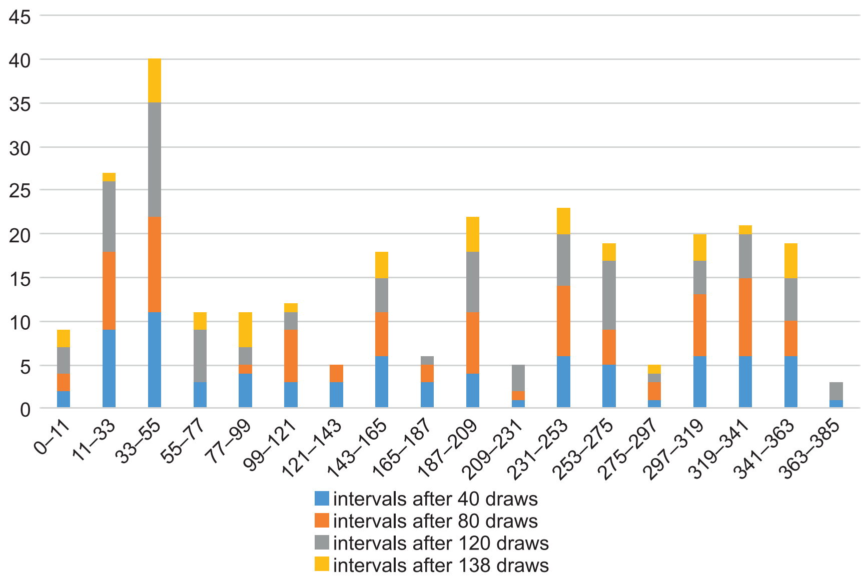


Fig. 6. Interval frequency histogram from generated numbers

As in the first example, we can observe a large variation between the counts of the given intervals (see Fig. 6). While presenting this graph lets us notice, some intervals have zero count after a given number of generator's passes. It can be seen, for example, in the range from 121 to 143 where only values after 40 and 80 generations appeared or in the last range where we can see occurrences after 40 and after 120 generations. The most even distribution between the intervals is in the case of the first series, where we can see small differences between the intervals.

4. Example 3

Input message: GujRplS53

$j = 2$

$k = 5$

$m = 4213$

$n = 462$

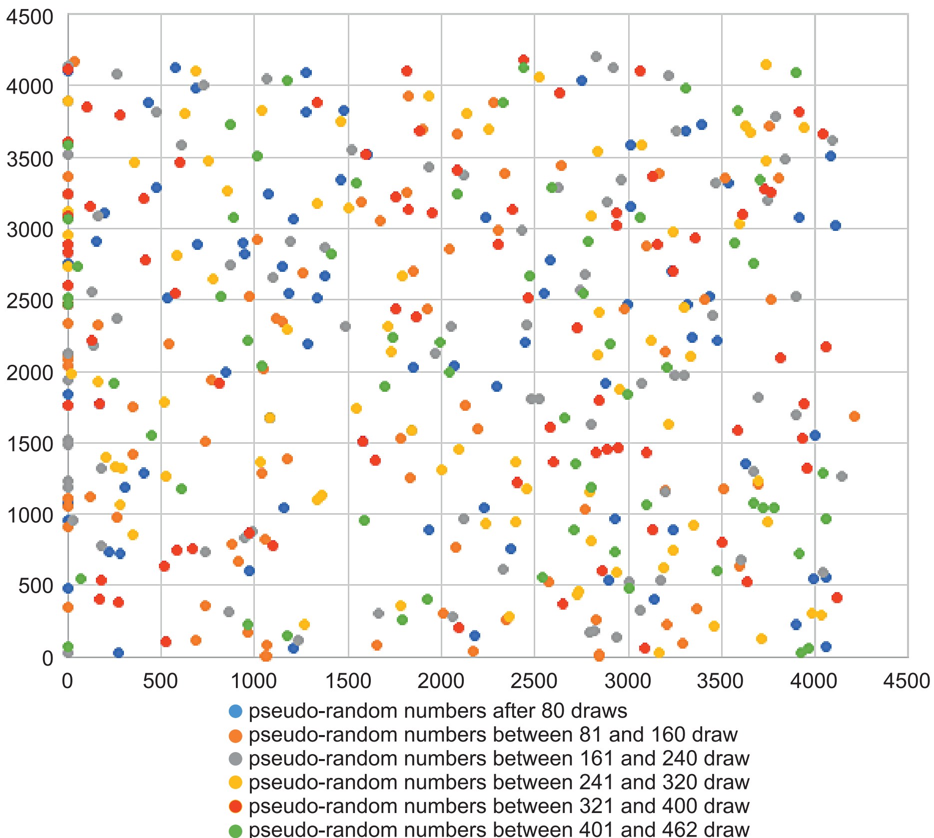


Fig. 7. Plot of distribution pseudo-random numbers A on the x-axis and pseudo-random numbers B on the y-axis

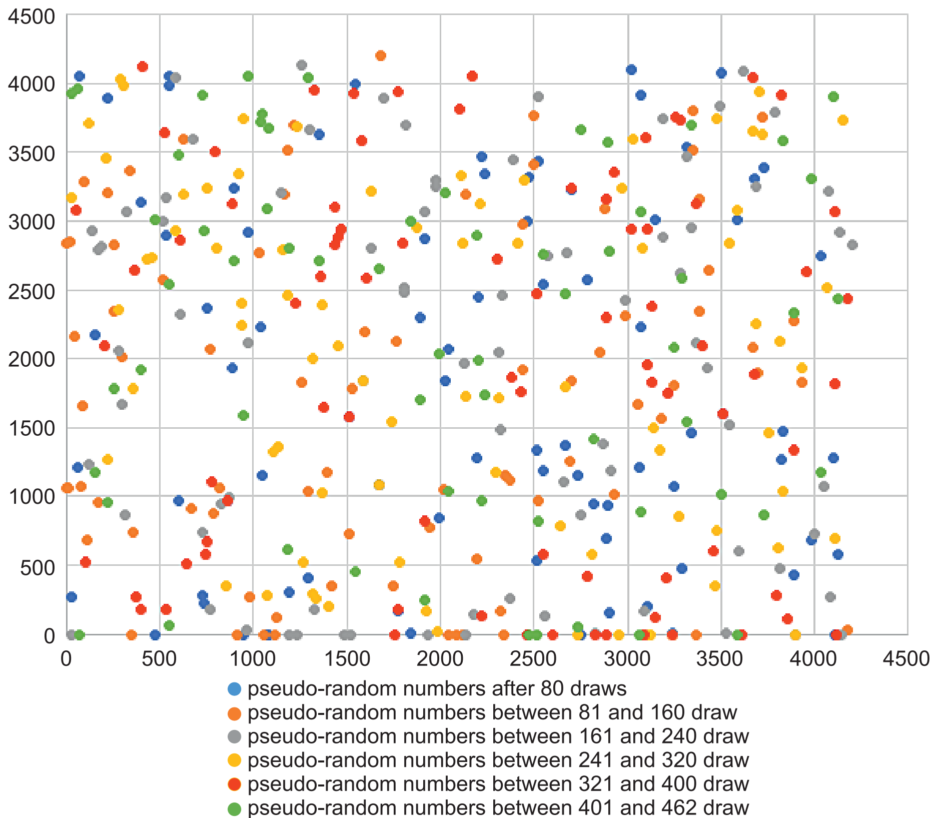


Fig. 8. Plot of distribution pseudo-random numbers B on the x-axis and pseudo-random numbers A on the y-axis

The graphs (see Figs. 7, 8) of the third example show the distribution of a large number of points, which have been divided into six insertion series, each series containing 80 points after the last. Here, we can see that the points are evenly distributed over the whole area of the graph. As in the previous examples, there is a generator period, but its size is much larger than in the other two examples. The generator period for this example starts at 83 generations and ends at 462 generations. This means that at 463 generations the values from the 83rd generation will be repeated.

The graph (see Fig. 9) showing the distribution of the numbers in the intervals shows a good distribution between the individual intervals. The exception is the first interval, in which, as we observe on the graphs of distribution, the value 0 is repeated, which overestimates the size of the first interval. As an exception to the equal abundance between compartments we can also consider the last allocation, in which the maximum values occur. An even distribution of the

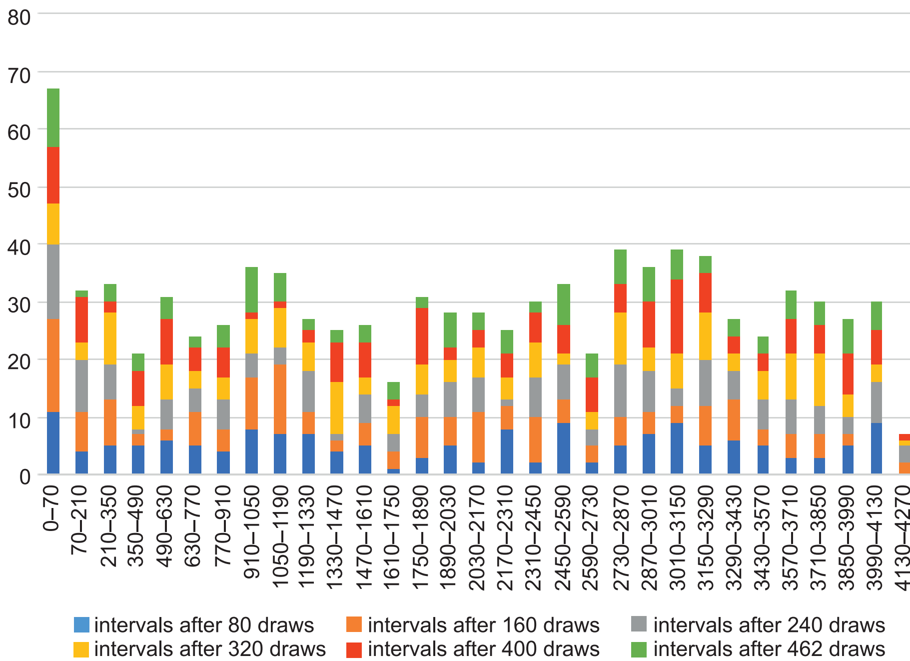


Fig. 9. Interval frequency histogram from generated numbers

abundance can also be observed if we look at the distribution between the series that was created when creating the graph.

5. Summary of examples

Summarizing the presented examples, we can observe that the values of pseudo-random numbers that will be created during the generation depend on the input parameters. During the generation we can get several generator periods. During the generation of numbers, there is a generator period only for the pseudo-random number B, which will cause that the points can be arranged in a certain characteristic way. Nevertheless, if there is no generator period for the number A, and we put the obtained numbers on the distribution graph we will get different points. It can be seen in the graphs of the first two examples. It also happens that the pseudo-random numbers A and B start to repeat after some time simultaneously, for example, as in the third example where there is a simultaneous generator period for both numbers.

Two parameters are of great importance when generating pseudo-random numbers: the seed and the parameter m. It is advisable that the parameter m is a large number, at most, it can be equal to 2147483646.

Let us now move on to testing the correctness of our technique.

Validation of our method

1. Chi-square test of compatibility (WOJTATOWICZ 1998)

It is one of the oldest statistical tests. Allowing us to test the hypothesis that a population has a certain type of distribution (described by a certain distribution in the form of a function), which be continuous or stepwise. The only limitation is that the sample be large, containing at least several dozen samples because the results be divided into certain classes of values. These classes should not be too few, at least 8 results should fall into each of them. In the case when there is a class smaller than 8 in the empirical distribution, this class should be combined with the neighboring one. For each class in the hypothetical distribution, theoretical counts are calculated, which are compared with the empirical counts using the appropriate chi-square statistic. When the discrepancies between the empirical and theoretical counts are too large, the hypothesis that the population has the assumed theoretical distribution be rejected.

The chi-square test formula: $\chi^2 = \sum_{i=1}^{k} \frac{(n_i - np_i)^2}{np_i}$.

The ranges were established in the same way as in the chart presentation. Results of the chi-square test for the examples shown.

Table 1

Chi-square test for the first example

For numbers	*K*	*R*	Degrees of freedom	Test statistic	Critical value
A and *B*	11	2	8	105.7046	15.5073
A	7	2	4	71.975	9.4877
B	6	2	3	8.3021	7.8147

Table 2

Chi-square for the second example

For numbers	*K*	*R*	Degrees of freedom	Test statistic	Critical value
A and *B*	13	2	10	172.5347	18.307
A	10	2	7	47.4164	14.0671
B	9	2	6	149,3395	12.5915

Table 3

Chi-square test for the third example

For numbers	*K*	*R*	Degrees of freedom	Test statistic	Critical value
A and *B*	30	2	27	762.6274	40.1132
A	21	2	18	546.9795	28.8692
B	22	2	19	149.4899	30.1435

The null hypothesis for each test performed was rejected. It can be noted that the B numbers from the test for the first example were closest to the normal distribution. For each test, a probability of 0.05 was used to create the critical value.

In the tables, the parameter k is defined as the number of intervals taken for the test, r as the number of parameters taken to calculate the normal distribution (in this case the mean and standard deviation of the generated numbers).

2. Kolmogorov's lambda consistency test (WOJTATOWICZ 1998)

In the Kolmogorov's lambda consistency test, to verify the hypothesis that the population has a certain type of distribution, one does not, as in the chi-square test, consider the counts of the empirical series and compare them with the counts of the hypothetical series, but compares the empirical and hypothetical distributions. Because when the population has a distribution consistent with the hypothesis, the values of empirical and hypothetical distributions should be similar at all tested points. The test begins by analysing the differences between the two distributions. The largest of which will then be used to construct the lambda statistic, whose distribution does not depend on the form of the hypothetical distribution. This distribution determines the critical values in this test.

The formula for calculating the value of a statistic for a given interval: $D = \sup|F_n(x) - F(x)|$

The formula for calculating the value of the test statistic: $\lambda = \max\left(D\sqrt{n}\right)$

For a fixed confidence level l we read the critical value from the limiting Kolmogorov distribution. The ranges were determined in the same way as in the presentation of the graphs.

Table 4

Kolmogorov test for the first example

For numbers	Alfa	Test statistic	Critical value
A and *B*	0.05	1.2914	1.36
A	0.05	8.4866	1.36
B	0.05	1.3687	1.36

Table 5

Kolmogorov test for the second example

For numbers	Alfa	Test statistic	Critical value
A and *B*	0.05	2.3528	1.36
A	0.05	1.6761	1.36
B	0.05	2.3269	1.36

Table 6

Kolmogorov test for the third example

For numbers	Alfa	Test statistic	Critical value
A and *B*	0.05	2.5839	1.36
A	0.05	2.1126	1.36
B	0.05	1.8966	1.36

The Kolmogorov test values are more approximate as to the critical value. The null hypothesis was not rejected for the numbers *A* and *B* from the first example, so we can conclude that the generated numbers have a normal distribution with the parameters of the mean and standard deviation.

3. A critical analysis of our method
A problem associated with the hybrid pseudo-random generator is the creation of short-range pseudo-random numbers. It is mainly due to the setting of the parameter *m* (responsible for the upper range of the generated values), such an example would be the value 100 or 768. In some cases a small adjustment of the value of *m* is enough to get better quality-generated numbers. A superb choice of *m* values are prime numbers, in their case it is rare to generate pseudo-random values with a small period. It is also not advisable to create a seed where a character repeats several times.

In the next section we test our method in practice by applying the drawn objects to the classification process.

Our method in action

The final part of verifying the performance of our technique was to perform classification using the kNN technique in the Monte Carlo Cross Validation model. We used an implementation in *R* language from the kNN library. Our goal was to check results of performed classification tests, in which objects

of training systems were generated by our method have distribution close to normal. The effect of the following test on the Statlog (Australian Credit Approval) Data Set (ICS-a. Online) and on the Statlog (Heart) Data Set (ICS-b. Online) is presented in the figures Interval frequency histogram from Australian Credit Approval Data Set and Interval frequency histogram from Heart Data Set.

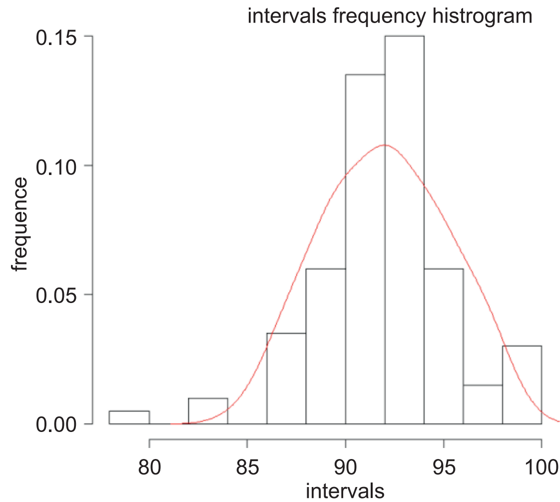


Fig. 10. Interval frequency histogram from Australian Credit Approval Data Set

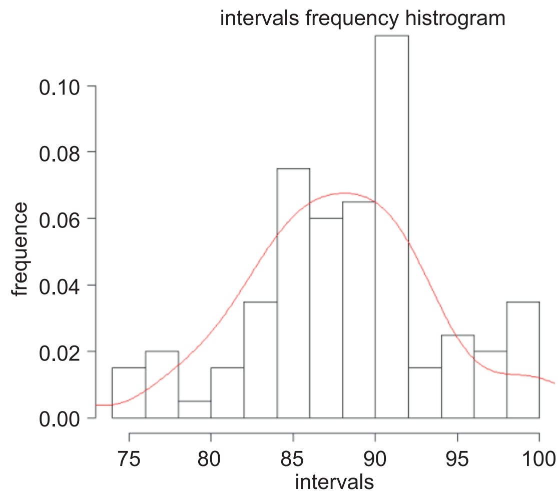


Fig. 11. Interval frequency histogram from Heart Data Set

Test Purpose

The focus of the test is the proportion of zeroes and ones for the entire sequence. The purpose of this test is to determine whether the number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence. The test assesses the closeness of the fraction of ones to ½, that is, the number of ones and zeroes in a sequence should be about the same. All subsequent tests depend on the passing of this test (RUKHIN et al. 2010).

Test description:

– conversion pseudo-random numbers to binaries and create binary string;

– conversion to ±1 (0 is converted to values -1, 1 is converted to values +1) and create sum S_n from binary string;

– compute the statistic: $S_{\text{obs}} = \frac{|S_n|}{\sqrt{n}}$ where n is length of binary string;

– compute P-value = $\text{erfc}\left(\frac{S_{\text{obs}}}{\sqrt{2}}\right)$, where erfc is the complementary error function.

If P-value is > 0.01, then conclude that the sequency is random.

For our tests binary string was created with first twenty generation from our examples use in section Proposed Methodology.

1. Test purpose for first example

$n = 515$
$S_n = 107$
$S_{\text{obs}} = 4.7149$
P-value = 2.4173e-06
Sequency is non-random

2. Test purpose for second example

$n = 295$
$S_n = 17$
$S_{\text{obs}} = 0.9897$
P-value = 0.3223
Sequency random

3. Test purpose for third example

$n = 434$
$S_n = 24$
$S_{\text{obs}} = 1.1520$
P-value = 0.2493
Sequency random

Conclusions

In this paper, we have presented a hybrid method of pseudo-random number generator, which is based on existing techniques for generating pseudo-random numbers. We have shown how to obtain pseudo-random numbers using the hybrid generator, examples with the distribution of numbers on graphs and graphs showing the counts of created values into intervals. In the presentation of the examples we showed its strengths and weaknesses, which include poor randomness in cases where we set a small value of the parameter m. As well as good points in the case of the third example where, despite the occurrence of a large period of the generator we see excellent randomness. The last stage that is presented is the tests of the correctness of the generator. If we looking on our examples in section test purpose we can see good quality of randomness in examples two and three. Only in first example created binary string is non--random. P-value for second example is 0.3223, for third example 0.2493 what can be mean that randomness for this generated numbers is very good quality. Only in first example created binary string is non-random.

References

AHRENS J.H., DIETER U. 1988. *Efficient table – free sampling methods for the exponential, Cauchy, and normal distributions*. Communication of the ACM, 31(11): 1330-1337.

BELL J.R. 1968. *Algorithm 334: Normal random deviates*. Communication of the ACM, 11(7): 498.

BOX G.E.P., MULLER M.E. 1958. *A note on the generation of random normal deviates*. Annals of Mathematical Statistics, 29(2): 610-611.

ICS-a. Donald Bren School of Information and Computer Sciences. University of California, Irvine. https://archive.ics.uci.edu/ml/machine-learning-databases/statlog/australian/australian.dat.

ICS-b. Donald Bren School of Information and Computer Sciences. University of California, Irvine. https://archive.ics.uci.edu/ml/machine-learning-databases/statlog/heart/heart.dat.

KINDERMAN A.J., MONAHAN J.F. 1977. *Computer generation of random variables using the ratio of uniform deviates*. ACM Transactions on Mathematical Software, 3(3): 257-260.

KNOP R. 1969. *Remark on Algorithm 334 [g5]: normal random deviates*. McGill University, Montreal.

Lagged Fibonacci Generator. Security and So Many Things, Asecuritysite. https://asecuritysite.com/encryption/fab.

RUKHIN A., SOTO J., NECHVATAL J., SMID M., BARKER E., LEIGH S., LEVENSON M., VANGEL M., BANKS D., HECKERT A., DRAY J., VO S. 2001. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD.

STALLINGS W. 2012. *Kryptografia i bezpieczeństwo sieci komputerowych – matematyka szyfrów i techniki kryptologii*. Helion, Gliwice.

SULEWSKI P. 2019. *Porównanie generatorów liczb pseudolosowych*. Wiadomości Statystyczne, 7: 5-31.

WIECZORKOWSKI R., ZIELINSKI R. 1997. *Komputerowe generatory liczb losowych*. Wydawnictwo Naukowo-Techniczne, Warszawa.

WOJTATOWICZ T.W. 1998. *Metody analizy danych doświadczalnych*. Wydawnictwo Politechniki Łódzkiej, Łódź.