Optimisation of Algorithms Generating Pseudorandom Integers with Binomial Distribution

Roman Horváth
Department of Mathematics and Computer Science, Faculty of Education,
Trnava University in Trnava, Trnava, Slovak Republic
roman.horvath@truni.sk

***Abstract*—This article summarises the creation of two approaches how to produce pseudorandom integers with binomial distribution and their comparison with other selected implementations. The first approach does the work by preparing a probability table searchable by the binary search algorithm, and the second one is about generating the values using the Galton board simulation. The second approach is slower (significantly slower for bigger numbers of trials) but is applicable in situations where the probability of success (connected to the binomial distribution process) is not uniformly distributed.**

I. Introduction

The motivation for this algorithm was seeking a proper generator for a simulation system that is currently in development. This simulation system is developed to be used in the educational process; however, its other use is not excluded either. Originally, a Poisson distribution was needed, but the binomial distribution came across as an algorithm that was easier to implement (especially the Galton board simulation variant), and that may produce Poisson-like results when proper parameters are passed to it [1, 2].

This is not the first time we deal with the algorithm's optimisation. Back in 2020, we published a paper at this same conference that announced an improved algorithm for calculating the distance between a point and a line [3]. This encouraged us to continue with such activities. Any small improvement is a contribution, especially when few people pay any attention to a specific area. Of course, it is ineffective to put energy into things nobody needs. In this case, we needed it, and we hope it helps anyone else. Thus, we will publish the algorithm in the public domain, like in the previous case.

All algorithms, including ours, are implemented in Java. We have used this programming language in the education process for several years (see, e.g., [4]), and we are sometimes inspired by the work of other colleagues in this area and connected areas (like [5, 6]).

II. The Binomial Distribution

The binomial distribution shows the probability of getting some result when some event occurs a specific number of times in case every single event is a result of two possible outcomes (for example, success or failure; hence the "binomial" in the name) [7, 8, 9].

The theoretical background is described on Wolfram's page [1]. According to the page: "The binomial distribution gives the discrete probability distribution P of obtaining exactly n successes out of N Bernoulli trials. (Where the result of each Bernoulli trial is true with probability p and false with probability $q = 1 - p$.)" The leaving part is thus the following formula:

$$P_p\,(n \mid N) = \binom{N}{n} p^n\, q^{N-n}.$$

For simplicity, we may, for example, explain P as the number of heads from flipping a fair coin n-time. Then the probabilities of getting head p or tail q of a fair coin toss are the same: 0.5.

We have found several implementations of the binomial distribution algorithm: Apache as part of Math3 library [10], CERN as part of Colt library [11], and as a part of SSJ (Stochastic Simulation in Java) library [12] that we compared with our implementation. The key input parameters for all implementations are n and p, which have the meaning of the number of trials and the probability of success (of a single trial). Each implementation supports passing a custom pseudorandom generator to the instance used to calculate the outcome (number of successes; and thus failures) after the n trials.

We had our own idea of the implementation in advance, which means before we started to search for the other implementations. All mentioned algorithms were released with some public domain licence, so we were able to investigate their code. We found out that all of them produce their results in real time by calculating the next generated value. We were headed in a different direction. Our algorithm was supposed to precalculate a table of probabilities and adapt it slightly that way so it would be searchable by the binary search algorithm. The table data adoption lies in creating increasingly arranged sums of probabilities (in a cumulative way) so that the data will represent "splits" that could be easily used to convert a uniform pseudorandom value (produced by a real number pseudorandom generator passed to the class instance during the construction) to a binomial pseudorandom integer.

III. The Preparation and Implementation

To prove that our algorithm is suitable for use in practice, we chose the following approach:

979-8-3503-2033-6/22/$31.00 ©2022 IEEE

1. Analyse existing algorithms.
2. Implement our algorithms (possibly with some variants).
3. Compare the speed performance of all algorithms.
4. Compare the distribution character of all algorithms.
5. Evaluate the results.

The algorithm was eventually implemented in two variants. One variant produces the binomial pseudorandom series using the uniform pseudorandom generator to get the probability of successes after n trials (just the resulting one) and a pre-calculated table for conversion. This is the faster approach. The other variant uses a non-uniform pseudorandom generator to get the probabilities of success for single trials; it gets the resulting integer using Galton board simulation and is slower in comparison to the first approach. The `BinomialDistribution` class chooses the variant internally using the `RandomGenerator` interface that declares the `isUniform` method that is used to make the decision.

The first variant, in summary, implements the idea that is, at its core, simple (create a table that would map the uniform pseudorandom real number to a bionomically distributed pseudorandom integer). Having that in mind, the path to the implementation was quite straightforward. The instance variables are p (probability of success), q (which is $1 - p$), n (number of trials) and the table of splits (array of doubles) calculated from probabilities that separate single outcomes. This is the map that converts a uniform pseudorandom (real) number to a binomial pseudorandom integer. The table of splits is precalculated using the following pseudocode:

```
array factorial[n + 1]
factorial[0] = factorial[1] = 1

for i = 2 to n
    factorial[i] = factorial[i - 1] * i

q = 1 - p
array table[n]

// Precompute the first value:
// Notes: 0! = 1; p⁰ = 1; so the
// quotient also is: (n! / n!) = 1;
// thus, the first value is: (1 - p)ⁿ
table[0] = q^n

i = 1
j = n - 1
while i < n
    denom = factorial[i] * factorial[j]
    quotient = factorial[n] / denom
    rest = p^i * q^j

    // Current probability adds to the
    // (sum of) previous ones to get
    // the continuous scale:
    table[i] = table[i - 1] +
        quotient * res
    ++i
    --j

for i = 0 to n - 1
    splits[i] = table[i]
```

The table is created using `BigDecimal` class (so the pre-calculation process was precise enough) and then converted to an array of primitive doubles. Then, when the class is asked to produce a binomial integer, the uniform real number pseudorandom value is generated, and this value is searched within the table of splits using a binary search algorithm. The position found in between the splits of the table is the generated integer. There are some corner cases to make quick decisions and to prevent some overflows, but the core is simply a binary search. This approach implies meeting one condition: the table must not be pre-calculating each time the binomial integer is generated. The class does the pre-calculation only during the initialisation process or after changing one of the parameters (p or n).

The second variant computes the resulting pseudorandom integer using the Galton board simulation (see, e.g. [13]). The principle is almost the same as used by BinomialConvolutionGen by SSJ [12]. The SSJ uses "the convolution method that generates Bernoulli random variates and adds them up." It sounds like a different approach, but if you look at the two implementations closely (ours and the SSJ's), you will find strong similarities. I believe that this is how you get the same (or at least similar) algorithm using different thinking. From the results, it looks like the Galton simulation algorithm (ours) is slightly faster than the convolution method (SSJ; the cumulative data shows a difference of about 0.1 microsecond), but this is negligible.

IV. The Test of Uniform Generators

All algorithms are based on the use of "third party" (in the meaning "outside the class" – in fact, it might be implemented by the same party) pseudorandom generators (presumably with uniform distribution) used to generate single "tosses" (trials) that determine the resulting binomial value. Each implementation came with its own uniform generator (hence the note about the "third party"), and we also used the standard Java implementation in the process. So following generators were considered:

- Apache – Well19937c generator [14] (henceforth referred to as "ApacheWell").
- CERN – MT19937 MersenneTwister [15] (henceforth referred to as "CernMerTwi").
- SSJ – MRG32k3a combined multiple recursive generator (CMRG) [16] (henceforth referred to as "SsjMrg").
- Standard Java Random class generator [17]. (See also: [18]; henceforth referred to as "JavaRandom.")

We are aware that the speed and quality of a specific uniform generator affects the results of the binomial generator, so we tested the speed of all uniform generators and selected two representatives to compare the speeds of all binomial distribution algorithms. (Note that all binomial classes support passing any uniform generator during the construction; possibly later too.) Table 1 compares the speeds of uniform generators after a hundred thousand operations. We measured and compared the performance of three instance methods: the one that generates the pseudorandom integers (`nextInt`), doubles (`nextDouble`), and longs (`nextLong`; if the generator supports it). The tests clearly show that the fastest generator comes from CERN (referred to as CernMerTwi). Therefore, the first choice was this generator. Other generators vary according to methods producing random numbers of different data types, so we

decided to use the default Java generator as the second one.

TABLE 1.
COMPARISON OF TIME PERFORMANCE OF FOUR ALGORITHMS USED IN FOCUSED IMPLEMENTATIONS.

time [ms]	nextInt	nextDouble	nextLong
ApacheWell	0.5158	2.8626	1.9630
CernMerTwi	0.3045	0.5966	0.5201
SsjMrg	1.4774	2.2272	-
JavaRandom	0.8264	2.5311	1.8659

V. THE TESTS OF BINOMIAL GENERATORS

A. *The Setup*

After that, we started to compare the binomial algorithms. In the first test (the speed test), we compared the first variant of our algorithm (the split table variant) with three implementations enumerated in the beginning (Apache [10], CERN [11], and SSJ [12]). We will use the following labels for single algorithms:

- the implementation in Math3 library – Apache [10]: ApaBin,
- the implementation in Colt library – CERN [11]: CernBin,
- the implementation in SSJ library [12]: SsjBin,
- and our implementation: OurBin.

The other variant (Galton board simulation) was compared only to the BinomialConvolutionGen by SSJ [12]. The labels for the two are as follows:

- the BinomialConvolutionGen (SSJ library) implementation [12]: SsjCon,
- the Galton board simulation: OurGal.

We arranged all six algorithms in the tables and graph (below) in the above order. We performed this first test using two selected uniform generators: CernMerTwi and JavaRandom. This test is hardware-dependent, so we performed the measurement on more than one machine.

After the first test, another test was performed: checking the characters of the distributions produced by all algorithms. We did that by producing and averaging the waste numbers of binomial values by all generators. Then we created a graph to visually compare the character of all produced binomial distributions. Seeing the data, we considered this test sufficient to prove that all produced distributions create the same character of data. We do not plan to use our algorithm in a safety-critical environment, so we did not perform any other tests. This test was purely algorithmic, and thus it was (and is) independent of hardware, so we ran it only on single hardware (machine).

B. *The Tests*

Originally, the speed tests were performed on three machines, but something went wrong with the tests on a third (oldest and slowest) machine, so the tests made on that machine were discharged. The results measured on the two remaining machines are in tables 2 and 3. Table 2 shows the statistical means and average durations of all binomial generators while producing values in different magnitudes of powers of tens using the CernMerTwi generator, and table 3 compares the results while the JavaRandom generator was used.

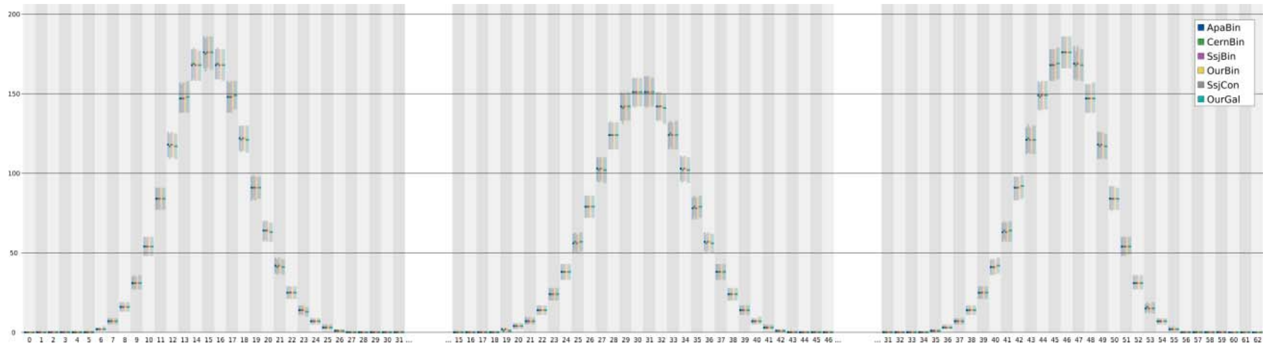
The second kind of test is supposed to show that all algorithms produce the same distribution. We wrote a simple testing application that was able to execute all generators a specified number of times (e.g., 1.500) with selected values of parameters: the probability of success (e.g., $p = 0.25, 0.5, 0.75...$) and the number of trials (e.g., $n = 10, 25, 30, 61...$). Then we created the number of

TABLE 2.
THE EXECUTION TIME (IN MILLISECONDS) OF POWERS OF TENS OPERATIONS MEASURED FOR ALL COMPARED ALGORITHMS USING THE CERN MERSENNE TWISTER MT19937 [15].

time [ms]	two runs on machine 1				two runs on machine 2			
	median	average	median	average	median	average	median	average
10ą repetitions (of generating 10,000 values)								
ApaBin	0.0400	0.0419	0.0365	0.0382	0.0293	0.0325	0.0298	0.0399
CernBin	0.0011	0.0016	0.0011	0.0016	0.0008	0.0013	0.0008	0.0016
SsjBin	0.0005	0.0006	0.0005	0.0006	0.0004	0.0005	0.0004	0.0006
OurBin	0.0004	0.0005	0.0004	0.0005	0.0003	0.0004	0.0003	0.0005
SsjCon	0.0025	0.0027	0.0025	0.0027	0.0018	0.0021	0.0018	0.0026
OurGal	0.0024	0.0026	0.0024	0.0026	0.0018	0.0021	0.0018	0.0025
10˛ repetitions (of generating 10,000 values)								
ApaBin	0.3807	0.3825	0.3450	0.3469	0.2729	0.2848	0.2737	0.2878
CernBin	0.0104	0.0104	0.0103	0.0104	0.0075	0.0077	0.0076	0.0079
SsjBin	0.0042	0.0042	0.0040	0.0041	0.0029	0.0030	0.0029	0.0030
OurBin	0.0039	0.0040	0.0039	0.0040	0.0028	0.0029	0.0028	0.0030
SsjCon	0.0243	0.0244	0.0241	0.0243	0.0172	0.0178	0.0173	0.0182
OurGal	0.0237	0.0238	0.0236	0.0238	0.0170	0.0178	0.0173	0.0181
10ł repetitions (of generating 10,000 values)								
ApaBin	3.8124	3.8266	3.4520	3.4651	2.7529	3.0532	2.7961	3.5228
CernBin	0.1004	0.1008	0.1007	0.1011	0.0740	0.0817	0.0738	0.0934
SsjBin	0.0410	0.0411	0.0412	0.0414	0.0317	0.0353	0.0286	0.0365
OurBin	0.0383	0.0385	0.0371	0.0374	0.0267	0.0297	0.0280	0.0358
SsjCon	0.2432	0.2439	0.2433	0.2441	0.1729	0.1916	0.1729	0.2214
OurGal	0.2358	0.2376	0.2355	0.2372	0.1718	0.1897	0.1730	0.2216
10 repetitions (of generating 10,000 values)								
ApaBin	38.2117	38.2524	34.5804	34.6441	28.8365	41.0302	29.3039	43.0858
CernBin	0.9843	0.9860	0.9892	0.9910	0.7640	1.1075	0.7728	1.1258
SsjBin	0.4022	0.4032	0.4080	0.4092	0.2901	0.4282	0.2974	0.4431
OurBin	0.3790	0.3799	0.3681	0.3692	0.2692	0.3969	0.2866	0.4280
SsjCon	2.4347	2.4385	2.4352	2.4398	1.8139	2.5747	1.8589	2.7191
OurGal	2.3664	2.3723	2.3632	2.3694	1.7773	2.5541	1.8591	2.7085

TABLE 3.
THE EXECUTION TIME (IN MILLISECONDS) OF POWERS OF TENS OPERATIONS MEASURED FOR ALL COMPARED ALGORITHMS USING THE STANDARD JAVA GENERATOR [17, 18].

time [ms]	two runs on machine 1				two runs on machine 2			
	median	average	median	average	median	average	median	average
10ą repetitions (of generating 10,000 values)								
ApaBin	0.0401	0.0416	0.0400	0.0414	0.0293	0.0324	0.0293	0.0325
CernBin	0.0012	0.0014	0.0012	0.0013	0.0009	0.0010	0.0009	0.0011
SsjBin	0.0006	0.0007	0.0006	0.0007	0.0004	0.0006	0.0004	0.0006
OurBin	0.0005	0.0006	0.0006	0.0006	0.0004	0.0005	0.0004	0.0005
SsjCon	0.0135	0.0138	0.0135	0.0138	0.0097	0.0106	0.0097	0.0106
OurGal	0.0136	0.0138	0.0135	0.0137	0.0098	0.0106	0.0098	0.0106
10˛ repetitions (of generating 10,000 values)								
ApaBin	0.3820	0.3841	0.3811	0.3830	0.2712	0.2840	0.2721	0.2846
CernBin	0.0114	0.0115	0.0114	0.0115	0.0080	0.0083	0.0080	0.0083
SsjBin	0.0054	0.0054	0.0055	0.0055	0.0038	0.0040	0.0038	0.0040
OurBin	0.0050	0.0050	0.0051	0.0051	0.0035	0.0036	0.0035	0.0037
SsjCon	0.1342	0.1349	0.1342	0.1349	0.0945	0.0990	0.0967	0.0997
OurGal	0.1352	0.1360	0.1343	0.1349	0.0953	0.0998	0.0974	0.1005
10ł repetitions (of generating 10,000 values)								
ApaBin	3.8246	3.8383	3.8161	3.8300	2.7046	2.8165	2.7362	3.3697
CernBin	0.1126	0.1130	0.1113	0.1116	0.0780	0.0807	0.0791	0.0975
SsjBin	0.0545	0.0547	0.0551	0.0553	0.0374	0.0388	0.0387	0.0481
OurBin	0.0501	0.0503	0.0497	0.0499	0.0341	0.0353	0.0352	0.0438
SsjCon	1.3435	1.3475	1.3433	1.3470	0.9450	0.9815	0.9451	1.1819
OurGal	1.3540	1.3577	1.3449	1.3486	0.9525	0.9903	0.9527	1.1909
10 repetitions (of generating 10,000 values)								
ApaBin	38.3289	38.3759	38.2248	38.3050	28.5546	38.7209	29.1498	42.4460
CernBin	1.1093	1.1113	1.0971	1.0997	0.7938	1.1037	0.8354	1.2151
SsjBin	0.5259	0.5272	0.5400	0.5426	0.3797	0.5319	0.4014	0.5914
OurBin	0.4856	0.4869	0.4744	0.4763	0.3452	0.4843	0.3613	0.5327
SsjCon	13.4564	13.4704	13.4564	13.4771	9.9055	13.5042	10.0366	14.8562
OurGal	13.5635	13.5775	13.4773	13.4986	10.0234	13.6301	10.2116	15.0113



Graph 1. Example of the runs using $n = 61$ and $p = 0.25$ (left), $p = 0.5$ (middle), and $p = 0.75$ (right). (Single run is computed from 1.000 distributions, each created by generating 1.500 values.)

"batches" of those runs and calculated the three average values of the data: overall average, an average of upper bound, and an average of lower bound; to smooth the curves and get the variations of the data. These tests showed that all algorithms positively produce the same distribution. The application was also able to draw and export graphs of all kinds. Examples of the runs are in graph 1.

VI. The Poisson Approximation

The initial motivation to implement our algorithm went through an attempt to get around implementing the Poisson distribution algorithm by implementing the simple version of the Galton board simulation. It produces binomial distribution convertible to an approximation of a Poisson distribution. Eventually, we created the table of splits algorithm that computes the table using a more complex implementation, but that does not mean that the original idea cannot be applied anymore.

According to [1, 2]: The binomial distribution converges towards the Poisson distribution with mean λ as the number of trials goes to infinity ($N \rightarrow \infty$) while the product $n.p$ remains fixed, or at least p tends to zero ($p \rightarrow 0$). Therefore, the Poisson distribution with parameter $\lambda = n.p$ can be used as an approximation to $B(n, p)$ of the binomial distribution if n is sufficiently large and p is sufficiently small. According to the two rules of thumb, this approximation is good if (for example) $n \geq 20$ and $p \leq 0.05$, or if $n \geq 100$ and $n.p \leq 10$.

VII. Conclusion

After performing all the tests and evaluating the measured results, we can conclude that our solution is applicable in practice. Our approach is faster than selected algorithms (if you do not force the instances to re-initialize their tables too often) and produces the distribution of the same quality. The first next step is to include the algorithm in the existing framework [19, 20] and then use it in our simulation system that is currently in development. The simulation system is developed with the intention to use it in the educational process (as a kind of educational material) as we do regularly with other systems and software [21, 22, 23] at our department. Still, any other use is not excluded either.

Acknowledgement

The work has been supported by the Cultural and Educational Grant Agency of the Ministry of Education, Science, Research and Sport of the Slovak Republic (KEGA) and the contribution was elaborated as part of the following KEGA projects: KEGA 013TTU-4/2021 entitled *Interactive animation and simulation models for deep learning* and KEGA 012TTU-4/2021 entitled *Integration of the usage of distance learning processes and the creation of electronic teaching materials into the education of future teachers.*

References

[1] *Binomial Distribution.* From: Wolfram MathWorld. Available at: ⟨https://mathworld.wolfram.com/BinomialDistribution.html⟩. Last accessed: 2022-09-16.

[2] (2012). *Binomial distribution.* From: Wikidoc by user WikiBot based on work by Brian Blank. Available at: ⟨https://www.wikidoc.org/index.php/Binomial_distribution⟩. Last accessed: 2022-09-16.

[3] Horváth, Roman – Fialová, Jana. (2020). The Creation of Simulation with an Algorithm Optimisation in Java for the Teaching Process. In *18th IEEE International Conference on Emerging eLearning Technologies and Applications (ICETA) : Information and communication technologies in learning.* Košice (Slovak Republic), Denver (USA) : Institute of Electrical and Electronics Engineers. ⟨https://doi.org/10.1109/ICETA51985.2020.9379150⟩. ISBN 978-0-7381-2366-0, pp. 160–166.

[4] Horváth, Roman. (2018). The Past Seven Years of Development of the Framework for Teaching Programming and the Students' Results. In *ICETA 2018 : 16th IEEE International Conference on Emerging eLearning Technologies and Applications : proceedings.* New Jersey (USA) : Institute of Electrical and Electronics Engineers. ISBN 978-1-5386-7912-8, pp. 185–189.

[5] Rokhmawati, Retno Indah – Az-zahra, Hanifah Muslimah. (2019). Identifying Students' Mental Model for Java Programming Subject. In *Proceedings of the 2019 3rd International Conference on Education and Multimedia Technology (ICEMT 2019).* New York, NY (USA) : Association for Computing Machinery, pp. 165–169. ⟨https://doi.org/10.1145/3345120.3345146⟩.

[6] Beňo, Pavel – Havan, Patrik – Šprinková, Sandra. (2020). Structured, Analytical and Critical Thinking in the Educational Process of Future Teachers. In *Acta Educationis Generalis.* Dubnica nad Váhom (Slovak Republic) : DTI University. ⟨https://doi.org/10.2478/atd-2020-0024⟩. ISSN 2585-741X. ISSN (online) 2585-7444. Volume 10, Issue 3, pp. 111–118.

[7] LaMorte, Wayne W. (2016). *The Binomial Distribution : A Probability Model for a Discrete Outcome.* Boston University School of Public Health. Available at: ⟨https://sphweb.bumc.bu.edu/otlt/mph-modules/bs/bs704_probability/bs704_probability7.html⟩. Last accessed: 2022-09-16.

[8] Frost, Jim. (2022). *Binomial Distribution : Uses, Calculator & Formula – Statistics by Jim.* Available at: ⟨https://statisticsbyjim

.com/probability/binomial-distribution/⟩. Last accessed: 2022-09-16.

[9] Glen, Stephanie. *Binomial Distribution : Formula, What it is, How to use it.* StatisticsHowTo.com : Elementary Statistics for the rest of us! Available at: ⟨https://www.statisticshowto.com/probability-and-statistics/binomial-theorem/binomial-distribution-formula/⟩. Last accessed: 2022-09-16.

[10] (2016). *Math – Commons Math : The Apache Commons Mathematics Library.* The Apache Software Foundation. Available at: ⟨https://commons.apache.org/proper/commons-math/index.html⟩. Last accessed: 2022-09-16.

[11] (2022). *Colt – CERN Open Source Libraries for High Performance Scientific and Technical Computing in Java.* CERN – European Organization for Nuclear Research. Available at: ⟨https://dst.lbl.gov/ACSSoftware/colt/, https://github.com/jenetics/colt⟩. Last accessed: 2022-09-16.

[12] L'Ecuyer, Pierre. (2018). *Stochastic Simulation in Java : SSJ.* SSJ User's Guide and Releases. Aisenstadt, Université de Montréal, Québec, Canada. Available at: ⟨http://umontreal-simul.github.io/ssj/docs/master/, https://github.com/umontreal-simul/ssj/releases⟩. Last accessed: 2022-09-16.

[13] *Galton Board.* From: Four Pines Publishing, Inc. Available at: ⟨https://galtonboard.com/⟩. Last accessed: 2022-09-16.

[14] Panneton, François – L'Ecuyer, Pierre – Matsumoto, Makoto. (2006). *Improved Long-Period Generators Based on Linear Recurrences Modulo 2.* ACM Transactions on Mathematical Software, volume 32, issue 1. Available at: ⟨http://www.iro.umontreal.ca/~lecuyer/myftp/papers/wellrng.pdf⟩. Last accessed: 2022-09-16.

[15] Matsumoto, Makoto – Nishimura, Takuji. (1998). *Mersenne Twister : A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator.* ACM Transactions on Modeling and Computer Simulation, volume 8, issue 1, pp. 3–30. Available at: ⟨https://dl.acm.org/doi/10.1145/272991.272995⟩. Last accessed: 2022-09-16.

[16] L'Ecuyer, Pierre. (1999). *Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators.* Operations Research, volume 47, issue 1, pp. 159–164.

[17] Knuth, Donald. (1998). *The Art of Computer Programming.* Stanford University, volume 2, section 3.2.1. Available at: ⟨https://doc.lagout.org/science/0_Computer%20Science/2_Algorithms/The%20Art%20of%20Computer%20Programming/%20%28vol.%202_%20Seminumerical%20Algorithms%29%20%283rd%20ed.%29%20%5BKnuth%201997-11-14%5D.pdf, https://seriouscomputerist.atariverse.com/media/pdf/book/Art%20of%20Computer%20Programming%20-%20Volume%202%20(Seminumerical%20Algorithms).pdf⟩. Last accessed: 2022-09-16.

[18] *Random (Java Platform SE 8).* Java™ Platform Standard Ed. 8 Documentation. 1993, 2022, Oracle and/or its affiliates. Available at: ⟨https://docs.oracle.com/javase/8/docs/api/java/util/Random.html⟩. Last accessed: 2022-09-16.

[19] Horváth, Roman. (2022). *GRobot programming framework documentation.* Available at: ⟨https://pdfweb.truni.sk/horvath/GRobot/⟩. Last accessed: 2022-09-16.

[20] Horváth, Roman. (2022). *GitHub – raubirius/GRobot : Contains the files of the Programming framework GRobot.* Available at: ⟨https://github.com/raubirius/GRobot⟩. Last accessed: 2022-09-16.

[21] Pokorný, Milan – Holý, Dušan. (2018). Interactive elements for teaching addition and subtraction. In *EME2018 Proceedings : Perspectives of primary mathematics education : 23rd scientific conference with international participation Elementary Mathematics Education.* Olomouc (Czech Republic) : Palacký University Olomouc. ISBN 978-80-905281-7-8, pp. 105–106.

[22] Štrbo, Milan. (2020). AI-based Smart Teaching Process During the COVID-19 Pandemic. In *Proceedings of the Third International Conference on Intelligent Sustainable Systems [ICISS 2020].* Piscataway (USA) : Institute of Electrical and Electronics Engineers. ISBN 978-1-7281-7089-3, pp. 402–406.

[23] Pokorný, Milan. (2021). Interactive Applications as an Additional Study Material for Teaching Mathematics at Secondary School During the COVID-19 Pandemics. In *UNINFOS 2021.* Žilina (Slovak Republic) : University of Žilina. ISBN 978-80-554-1828-5, pp. 34–38.