

QUALITY RANDOM NUMBER GENERATOR

Maria Dima^{a,*} Mihai-Tiberiu Dima^a, Svetlana Dima^a, Madalina Mihailescu^b

^a*Dzhelepov Laboratory of Nuclear Problems, JINR, RU-141980, Dubna, Russia,*

e-mail: mmdima@jinr.ru

^b*Hyperion University, Calea Calarasi 169, RO-030615. Bucharest, Romania*

Abstract – Numerous applications in physics and technology rely on random number generation: for Monte Carlo purposes, key distribution, and other tasks. For these elaborate hash functions with carefully studied and tuned algorithms have been developed, giving pseudo-random numbers. Depending on the complexity and quality of their output, they vary from very good quality (such as RANLUX with a 10^{171} repetition period), to fast algorithms, however of lesser period (such as the Mersenne Twister, a factor of ca. $\times 40$ faster). We here present the implementation of a true-random number “multiplier” algorithm. The algorithm relies on a finite set of true-random numbers from a physical source (in our case 0.2M atmospheric noise random numbers in the range of 0 ... 9999). The algorithm produces new numbers by combining pairs of 2 random numbers from the list, situated at random distance apart. The random offset is calculated by a shift register structure involving both the local rand() generator, and numbers from the list itself, whereby it produces "non-repetitive repetitions" - i.e. our multiplier has no known period. The tests, performed with the DieHarder [1] test suite, show good quality.

INTRODUCTION

In performing Monte Carlo simulations in physics [2, 3, 4] it is imperative to assure the good quality of the random numbers generated. Likewise, cryptography [5] depends crucially on what has come to be known as the “One Time Pad” [6] principle, requiring also true-random quality. Hash function based generators have a set of problems [7, 8], ranging from leak of distribution uniformity, neighbor correlation and seed states not having the same period, to problems with dimensional distribution. For example, in Fig.1 it's shown how the rand() from GCC (which is essentially a hash function) is having biases while generating random numbers - the errors for a helix fit on generated points. Similarly, sets of ISAJET-7.0 SUSY events [14] needed to be discarded due to like biases in their generation. Some of the most popular methods for random and pseudo-random number generators are: LCGs, Mersenne Twister and CSPRNGs and TRNGs.

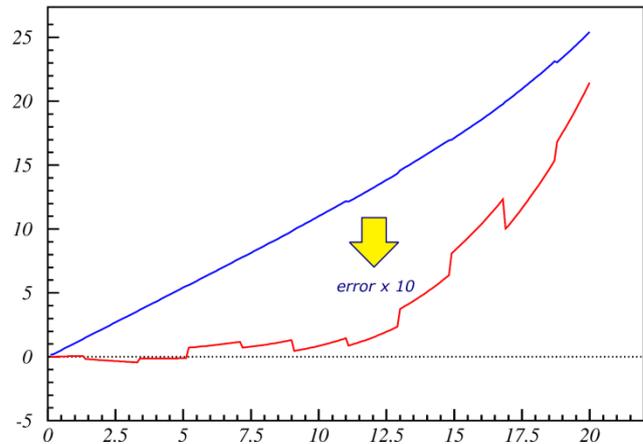


Figure 1 – helix fit errors [13] on generated points with Monte Carlo modelled measurement errors. Statistical error is in blue and the systematic error in red, magnified $\times 10$.

Linear Congruential Generators (LCGs) are simple pseudo-random algorithms using a linear recurrence relation: $X_{n+1} = (aX_n + c) \bmod m$. They are fast, memory-efficient, and

reproducible with a known seed, making them popular in simulations and basic applications. However, their periodicity and predictability (due to structured output patterns) render them unsuitable for cryptography. LCGs are historically significant, used in early programming languages like C and Java. **Mersenne Twister** is a pseudo-random algorithm, based on a twisted generalized feedback shift register, it boasts an extremely long period ($2^{19937} - 1$) and high-dimensional equidistribution. It balances speed and statistical quality, making it a standard in Python, R, and gaming engines for simulations and Monte Carlo methods. While not cryptographically secure, its robustness against most biases ensures reliability in non-security contexts like procedural generation or randomized algorithms. **Cryptographically Secure PRNGs** (CSPRNGs), such as AES-CTR or algorithms like Fortuna, combine pseudo-random techniques with cryptographic primitives to produce unpredictable outputs resistant to reverse-engineering. They often seed from TRNGs and use hashing or block ciphers to ensure security. Though computationally heavier, they are essential for encryption, token generation, and secure communications. Examples include Linux’s `/dev/urandom` and libraries like OpenSSL. **True Random Number Generators** (TRNGs) generate numbers from physical phenomena like electronic noise, radioactive decay, or quantum effects, ensuring true randomness. These methods are non-deterministic, offering high unpredictability, making them ideal for cryptography and security applications. TRNGs are valued in scenarios where absolute randomness is critical, such as encryption keys or scientific simulations (e.g. Monte Carlo).

We here present a portable random generator that uses a set of 0.2M true-random numbers, from atmospheric noise [9]. Due to the volume limitation of this set, we devised a method to enhance its volume to 20000M, which we term “multiplier”, that preserves the true-random quality of the set. We tested our algorithm with the DieHarder test suite (official test suite of the Swiss Federal Office for Metrology - METAS [10]) and compared it to the performance of the `gcc rand()` server.

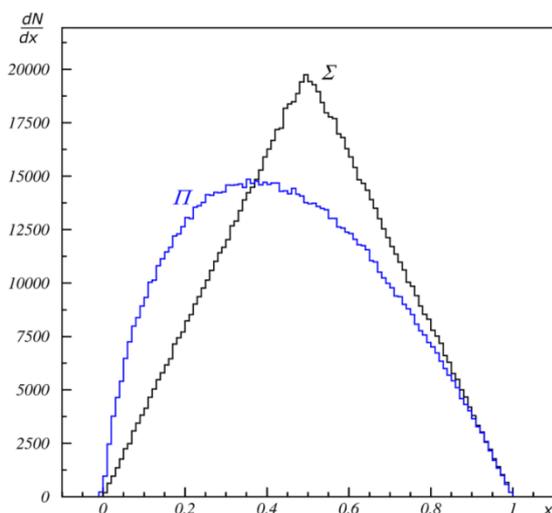


Figure 2 – distribution of random numbers generated by summing (black) or multiplying (blue) pairs of 2 numbers from the true-random set.

Do note here that our “multiplier” is dissimilar to Intel’s RdRand [11] server, which is a pseudo-random generator (AES-CTR-DRBG [12]) seeded at approximately 1 M/s from an on-chip IP-core entropy source.

Our method combines random pairs found at random distance apart, and does not use a hash function. Aside this, we do not rely on chip-dependent entropy sources, which vary (or are absent) depending on the manufacturer.

NEW METHOD USING ATMOSPHERIC NOISE

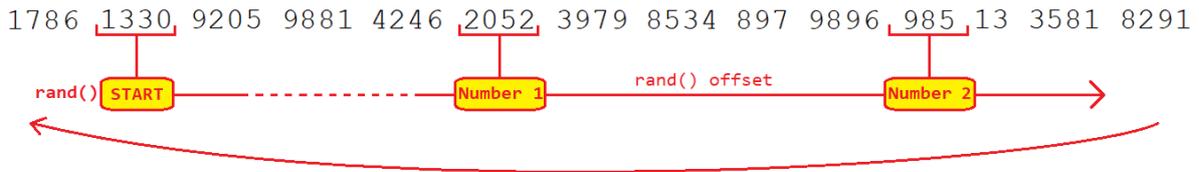


Figure 3 – Our method combines the digits of the 2 true-randoms with order given by gcc’s rand().

To address these shortcomings, we turned to a true-random source, from `random.org` [9], namely atmospheric noise. We downloaded sets of 10000 numbers, up to a total of 200k numbers, in the range of $[0 \dots 9999]$. This is however a very limited number and repetitions occur often. To enhance our source-volume, we combined true-random numbers from the list. This is, however, not that simple. While retaining “local” randomness, trivial combinations, such as sum and product (figure 2), distort the phase space and thereby affect the density.

A more fitting approach is to combine the digits of the two true-randoms in an order supplied by gcc’s rand(). After this step we norm the distribution to $[0 \dots 1]$. This approach gives us access to all $C_n^2 = 2 \cdot 10^{10}$ combinations, thereby solving the volume problem.

However, obtaining pairs from the list with a uniform long- and short-range coverage remains still unaddressed. For this we use a shift register structure of offsets. The first offset is generated by rand(); at the next serve it is passed onto the second offset, which is passed to the third, ... up to 10th. Each offset is divided by a prime number, of different size, such that the sum of the modulo’s of all offsets to their respective prime numbers covers uniformly the list in both long- and short-range. The last offset that remains empty is filled again by rand(). Additionally we add to the offset the current value of the random number in the box. The offset is typically beyond the list’s end, so thus we wrap around to the beginning (the number of times that it is needed). This is an elaborate method of using rand() to indicate access to any number in the list on a uniform long- and short-range scale.

It is important to note here that we need neither seeding, nor initialisation of our algorithm. We also could simply have taken system-time instead of rand() – which is an idea for further development, to make the server even faster (current throughput is 0.15 Gbps, significantly below Intel’s 3.00 Gbps server, however better than some of the established servers – most, if not all, replaceable by Intel RdRand [11] in both speed and randomness quality).

TESTS OF OUR METHOD

As aforementioned we used the DieHarder test suite for testing our algorithm. (For future studies we could benchmark it with NIST SP 800-22 also.) Figure 5-left shows that the distribution is as expected, flat in the $[0 \dots 1]$ interval. This remains true at all scales plotted, the rms of the data following a free-scale model.

phase space without distortion.

Not appealing to any has function, and just relying on a true-random physical source, sets our algorithm in the same class with other physical-source servers (for instance in the quantum-quality range, ID Quantique's Quantis [16], a ca. 3000€ server). This indicates that true-random servers are not quite in the same class as open-software.

CONCLUSION

We presented a non-hash function true-random server based on a physical source (atmospheric noise). The limited source-volume problem we solved by combining pairs of random numbers in the set. The even long- and short-range distribution of the pairs we solved with a displacement shift-register structure. The algorithm passed distribution, Fourier and DieHarder battery tests very well. It requires no initialization and offers a decent 0.15 Gbps throughput. It has a speed of approx. 180ns/number, compared with other true random number generators that (depending on their implementation and hardware quality), can range between 100ns up to several μ s, making our software to be between the fastest generators. Our method is more applicable than standard PRNs because in Monte Carlo simulations (for instance, not throwing away events that were generated badly), quantum simulation, testing software (fuzz testing methods), cryptography.

REFERENCES

1. DieHarder test suite: <https://github.com/GINARTeam/Diehard-statistical-test>
2. M. Stipčević, C. K. Koç, True Random Number Generators, Open Problems in Mathematics and Computational Science (2014): DOI:10.1007/978-3-319-10683-0_12
3. V. Mannalath, S. Mishra, A. Pathak, A Comprehensive Review of Quantum Random Number Generators: Concepts, Classification and the Origin of Randomness, Quantum Information Processing, 22, Article number: 439 (2023)
4. M. Moeini, M. Akbari, M. Mirsadeghi, H. R. Naeij, N. Haghkish, A. Hayeri, M. Malekian, Quantum Random Number Generator Based on LED, Journal of Applied Physics, Volume 135, Issue 8 (2024)
5. Priyanka, I. Hussain, A. Khaliq, Random Number Generators and their Applications: A Review., (2019), 7. 1777-1781.
6. L. Thomas, One-Time Pad, In book: Trends in Data Protection and Encryption Technologies, (2023) DOI:10.1007/978-3-031-33386-6_1
7. P. L'Ecuyer, Random Number Generators and Empirical Tests, Monte Carlo and Quasi-Monte Carlo Methods 1996. Lecture Notes in Statistics, vol 127. Springer, New York, NY (1998)
8. L. Pasqualini, M. Parton, Pseudo Random Number Generation: a Reinforcement Learning approach, Procedia Computer Science, Volume 170, 2020, Pages 1122-1127, ISSN 1877-0509
9. Atmospheric Noise downloadable data: <https://random.org>
10. Swiss Federal Office for Metrology – METAS, Random Number Generation - Certification (passed Dieharder test): <https://www.idquantique.com/random-number-generation/certifications/>
11. Intel's RdRand server: <https://www.intel.com/content/www/us/en/partner/showcase/offering/a5b3b000000zqrAAA/xip8001b-true-random-number-generator-trng-ip-core.html>
12. Y. KIM, S. C. SEO, Efficient Implementation of AES and CTR_DRBG on 8-Bit AVR-Based Sensor Nodes, IEEE Access, vol. 9, pp. 30496-30510, (2021) DOI:10.1109/ACCESS.2021.3059623
13. M.O. Dima, M.T. Dima, M. Dima, M. Mihailescu, Flash-algorithm for helix fit, Nucleus-2024 conference, accepted for publication in Journal of Modern Physics E, DOI:10.1142/S0218301324410064
14. M. Dima, J. Barron, A. Johnson, L. Hamilton, U. Nauenberg, M. Route, D. Staszak, M. Stolte, and T. Turner, Mass determination method for the left and right selectron above production threshold, Phys. Rev. D, Vol. 65, Issue 7, p. 071701(R) (2002), DOI:10.1103/PhysRevD.65.071701
15. Dieharder manual - what is a 3D Sphere test & it can fail at one of RGB test even if it's perfect: <https://rurban.github.io/dieharder/manual/dieharder.pdf>
16. ID Quantique random number generator: <https://www.idquantique.com/random-number-generation/products/quantis-random-number-generator/>