Random Number Generators: Implementation and Statistical Analysis Using Dieharder

Kashev Dalmia
Email: dalmia3@illinois.edu

David Huang
Email: huang157@illinois.edu

William H. Sanders
Email: whs@illinois.edu

***Abstract*—We implement in efficient C++ several kinds of different modern pseudo-random number generators, or PRNGs; a Linear Congruential Generator, a Lagged Fibonacci Generator, an Xorshift type generator, and a Mersenne Twister Generator. We discuss the challenges in testing these kinds of PRNGs, and discuss one of the more common frameworks for testing these random number generators. Finally, we compare our results to what we expect from these implementations, as well as comment on the suitability of these PRNGs for use in simulation of systems.**

***Index Terms*—Random Number Generators, Psuedo Random Number Generators, PRNGs, Statistical Analysis, Dieharder**

I. Introduction

The applications of randomness are far reaching. From statistics to simulation, there is a large need for random number generators to perform quickly, generate seemingly random numbers, yet be reproducible in case something needs to be random yet reproducible. Thus, the creation of pseudo random number generators, or PRNGs, has become a large research area in modeling and computer simulation. In the field of analysis of computing systems, simulation is an extremely important part of the modeling process. Today's stochastic models become so complex that solving these models analytically becomes impossible. Simulations driven by randomness are common, and motivate the study in this paper.

A. Kinds of Random Number Generators

For the purposes of this paper, there are two kinds of Random Number Generators, or RNGs: Hardware RNGs, and Software RNGs. Hardware RNGs are used not only to output a random bit stream faster than a multipurpose CPU might be able to [1], [2], but also in order to gather entropy in a more unpredictable way, such as measuring thermal noise, detecting air moisture, or using unpredictable user interactions, such as keystrokes and mouse motion , to name a few. Often hardware RNGs are used to seed software RNGs. Readers interested in the seeding of RNGs with external entropy are encouraged to read [3].

Software RNGs, unless they draw upon these sources of entropy from nature for seeds, are in fact PRNGs, because the software that generates these sequences must be deterministic. In this paper, the term RNG will refer to software PRNGs, unless otherwise indicated. These software RNGs use mathematical models to take a finite amount of state and turn this into a sequence of seemingly random output, suitable for use in a variety of purposes. Though special purpose hardware RNG circuits may be fast, software RNGs are usually faster than repeatedly polling a sensor to gain outside entropy. They are also far more convenient to use, as all is required is a computer, rather than special purpose hardware.

Finally, there are several classes of software RNG based on quality of output. The highest class is those which are suited for cryptography. Generally, this includes passing the next bit test: that if k bits of a random sequence are known, then no polynomial time complexity algorithm should be able to guess the next bit [4]. These cryptographically secure random number generators tend to be slow. They are not designed to be run more than a few times in cryptography, which makes them unsuitable for use in applications in which speed is necessary, like simulation.

This paper deals with RNGs which are not of cryptographic quality, but are nonetheless useful. These RNGs combine speed with memory complexity and quality of output.

B. This Project

In this project, we study and implement a few common random number generators in C++, from those which have been used historically, to those which are most commonly in use today. We test our generators using the Dieharder test suite. Using Dieharder, we can push these RNGs to the point of failure, and therein see their flaws for various applications. We can also compare our results with the results of the RNGs built into the GNU Scientific Library, or GSL, and thus Dieharder. All of the software written for this project is available on Github [5].

First, we discuss the methods and challenges of testing random numbers in II. Then, we discussed the different classes of RNGs we study in III. Finally, we describe our software approach in IV and share our results in V.

II. Testing Random Number Generators

This section will describe the difficulties and challenges to testing random number generators, including the rationale behind p value analysis done in dieharder. We will heavily reference the Dieharder manual in this section. Describing how dieharder works is essential, and perhaps describing major important tests, especially the ones from diehard.

III. Classes of Random Number Generators

Here will will describe different kinds of random number generators, the math that makes them work, give examples of each kind, and finally, give their strengths and weaknesses.

A. Linear Congruential Generators

This section will describe LCGs. http://en.wikipedia.org/wiki/Linear_congruential_generator

this section should decribe variants, such as RANDU, also http://en.wikipedia.org/wiki/Lehmer_random_number_generator

B. Issues

LCGs have a fatal flaw, that is amplified by bad parameter choosing, like those in RANDU: their outputs fall in easily identifiable planes, as proved by Marsaglia [6].

C. Lagged Fibonacci Generators

This section will describe Lagged Fib Generators. http://en.wikipedia.org/wiki/Lagged_Fibonacci_generator Also Mention Subtract with Carry: http://en.wikipedia.org/wiki/Subtract_with_carry

D. Xorshift Generators

This section will describe Xorshift generators.

http://en.wikipedia.org/wiki/Xorshift

E. Mersenne Twister Generators

This section will describe the Mersenne Twister class of generators.

http://en.wikipedia.org/wiki/Mersenne_twister

IV. Software Implementation

Describe the software approach, the implemented generators, etc.

V. Results and Analysis

Describe the results of the dieharder tests for our implementations. Discuss speed, and suitability for simulation.

VI. Conclusion

Talk about the generators.

References

[1] C. Saiprasert, C.-S. Bouganis, and G. A. Constantinides, "An optimized hardware architecture of a multivariate gaussian random number generator," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 1, pp. 2:1–2:21, Dec. 2010. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/1857927.1857929

[2] M. Barel, "Fast hardware random number generator for the tausworthe sequence," in *Proceedings of the 16th Annual Symposium on Simulation*, ser. ANSS '83. Los Alamitos, CA, USA: IEEE Computer Society Press, 1983, pp. 121–135. [Online]. Available: http://dl.acm.org.proxy2.library.illinois.edu/citation.cfm?id=800042.801454

[3] C. Hennebert, H. Hossayni, and C. Lauradoux, "Entropy harvesting from physical sensors," in *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '13. New York, NY, USA: ACM, 2013, pp. 149–154. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/2462096.2462122

[4] A. C. Yao, *Theory and applications of trapdoor functions*, 1982, pp. 80–91.

[5] K. Dalmia. (2014) kashev/rngs: A project for ECE 541 project #2 @ UIUC. Investigating Random Number Generators. [Online]. Available: https://github.com/kashev/rngs

[6] G. Marsaglia, "Random numbers fall mainly in the planes," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 61, pp. 25–28, 1968.