Modern Random Number Generators: Implementation and Statistical Analysis

Kashev Dalmia
Email: dalmia3@illinois.edu

David Huang
Email: huang157@illinois.edu

William H. Sanders
Email: whs@illinois.edu

***Abstract*—We create several PRNGs and then test them using a common battery of statistical tests.**

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

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. Goals of the Project

We plan to create implementations of several random number generators, and then analyze their effectiveness via several different metrics. Our plan for implementing the PRNGs is in Section II, and our plan for the analysis of these generators is in Section III. We aim to create usable PRNGs, and then prove that they are usable as engines for simulations by passing common statistical tests.

II. Software Plan

We plan to use C++11 to create our PRNGs. C++11 introduces a `std::uniform_real_distribution` [1] which can be used to convert PRNG outputs into usable forms, as well as providing reference implementations for common PRNGs like the `std::mersenne_twister_engine`, based on the Mersenne twister algorithm [2]. We can both reproduce these algorithms in the C++11 STL, as well as algorithms not implemented in the STL. Our implementations can be designed as functors, then passed to `std::uniform_real_distribution` to create a uniform random variable. Alternatively, the generated bitstream, the random integers which are created by the generator, can be analyzed as well. These structures will be created within the GNU Scientific Library's class for random number generators, for portability.

Some of the generators we intend to implement are the following:

- Lehmer RNG [3]
- Mersenne Twister [2]
- WELL - Well Equidistributed Long-period Linear [4]

- Xorshift [5]
- ACORN [6]

These PRNGs vary in their complexity and suitability for complex applications. Some do not pass modern tests, but proving that they do not pass modern statistical analysis is still a worthwile exercise.

III. Analysis Plan

There are many ways to test the effectiveness of random number generators. We plan to use several different analysis methods. The first is the series of Diehard tests developed by George Marsaglia [7]. These tests can either be reproduced from their descriptions, or used directly. Related to these tests are the Die Harder tests. However, most valuable is a toolkit published by the National Institute of Standards and Technology on the testing of Random Number Generators [8]. This toolkit includes source code, as well as a detailed implementation guide.

The Die Harder suite, as well as the NIST toolkit, support the GNU Scientific Library wrapper for RNGs. Thus, structuring our code within the GSL structures will allow us to use the source to stream the output of our implementations of the RNGs to the existing tests.

IV. Anticipated Issues

With the guidance of the resources we've found, we don't anticipate many issues. One issue that we may encounter is the difficulty of plotting data in meaningful ways in C++. We can either use a GNUPlot library, or export the data to `.mat` files for plotting in MATLAB.

V. Conclusion

We anticipate enjoying this project and look forward to reproducing some interesting PRNGs.

References

[1] cppreference.com. (2014) std::uniform_real_distribution. [Online]. Available: http://en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution

[2] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998. [Online]. Available: http://doi.acm.org.proxy2.library.illinois.edu/10.1145/272991.272995

[3] W. H. Payne, J. R. Rabung, and T. P. Bogyo, "Coding the lehmer pseudo-random number generator," *Commun. ACM*, vol. 12, no. 2, pp. 85–86, Feb. 1969. [Online]. Available: http://doi.acm.org/10.1145/362848.362860

[4] F. Panneton, P. L'Ecuyer, and M. Matsumoto, "Improved long-period generators based on linear recurrences modulo 2," *ACM Trans. Math. Softw.*, vol. 32, no. 1, pp. 1–16, Mar. 2006. [Online]. Available: http://doi.acm.org/10.1145/1132973.1132974

[5] F. Panneton and P. L'ecuyer, "On the xorshift random number generators," *ACM Trans. Model. Comput. Simul.*, vol. 15, no. 4, pp. 346–361, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1113316.1113319

[6] R. S. Wikramaratna, "The additive congruential random number generator-a special case of a multiple recursive generator," *J. Comput. Appl. Math.*, vol. 216, no. 2, pp. 371–387, Jun. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.cam.2007.05.018

[7] G. Marsaglia. (1995) The marsaglia random number cdrom including the diehard battery of tests of randomness. [Online]. Available: http://www.stat.fsu.edu/pub/diehard/

[8] L. E. Bassham, III, A. L. Rukhin, J. Soto, J. R. Nechvatal, M. E. Smid, E. B. Barker, S. D. Leigh, M. Levenson, M. Vangel, D. L. Banks, N. A. Heckert, J. F. Dray, and S. Vo, "Sp 800-22 rev. 1a. a statistical test suite for random and pseudorandom number generators for cryptographic applications," Gaithersburg, MD, United States, Tech. Rep., 2010.