Evaluation of Common Pseudo-Random Number Generators

## **How does a pseudo-random number generator’s algorithm affect its quality?**

Julia Bowers

Computer Science

Supervisor: Jessica Patterson

[**Introduction**](#_cx4ls8f5jtep)2

[**Applications of Pseudo-Random Number Generators**](#_v8cu1wsxynue)3

[**Commonalities Among All Pseudo-Random Number Generators**](#_oxr97macmjfw)4

[**Pseudo-Random Number Generators of Interest**](#_4o1756ac48b5)6

[Linear Congruential Generator](#_kmunudkm620s) 6

[Mersenne Twister](#_lc136i1pen71) 7

[**Summary of Statistical Tests**](#_tpvfwcrruwj)8

[**Investigation**](#_2eytgad0y6fj)11

[Comparison of Linear Congruential Generators](#_i3aj6e508n3r) 11

[Comparison of best LCG, Mersenne Twister, SHA-1, and MRG](#_buka2t8ixery) 14

[Speed Test Results of 10^8 calls to each generator:](#_hbz2nwk07mkf) 15

[**Conclusion**](#_8doifyvkkby8)16

[**Appendixes**](#_5j9gdothaens)17

[Appendix A: C code used to run battery of statistical tests on PRNGS](#_q2ptqcmqh62) 17

[Appendix B: C code used to run time tests](#_w2ptkx74v5u0) 19

[Appendix C: Octave code to produce RANDU three-dimensional plot](#_cypkzmyq3zpn) 20

[Appendix D: Results of Statistical Tests on Linear Congruential Generators](#_i7o0lp1yp50l) 20

[Appendix E: Results of Statistical Tests on Various PRNGs](#_p1acmrbii051) 22

[**Bibliography**](#_6k7yp2sfxt3w)23

## 

## 

## Introduction

Randomness is essential to many applications, from simulations, to games, to secure internet connections. What does a computer do though, when it can’t flip a coin hundreds of times, to generate a unique random number? How can a deterministic machine be programmed to generate something “random”? To overcome this issue, computers are programmed to generate numbers that aren’t truly random (completely lacking in pattern and predictability), but rather only appear random, and are thus termed “pseudo”-random numbers.

In order to produce *true* random numbers, hardware random number generators measure stochastic physical processes, such as atmospheric noise or radioactive decay. However, it is often not practical to use hardware random number generators (RNGs) as they are slower, need additional physical devices, and the result is not reproducible. They can be useful in the generation of a seed (initial) value for another RNG however. Pseudo-random number generators (PRNGs), instead, are algorithms designed to produce either a random byte or floating point number uniformly distributed between 0 and 1 based on an initial seed value. With the same seed value and same algorithm parameters, the PRNG will output the same sequence.

We generally have an intuitive feeling of what randomness is, which makes a sequence such as 0011000001001011 feel more random than 0101010101010101, though both are equally likely to occur. Some definitions of randomness are based on lack of knowledge of initial conditions (from which randomness arises) or indifference to the next value of the sequence (lack of bias). This is called subjective randomness, as it depends on human interpretation and assessment. This sense of randomness is very difficult to code into machines, and it may not even be a good test of randomness. To most of us, even if the PRNG generation algorithm is known and every value is therefore calculable, even the poorest PRNGs that would result in less accurate results in application, appear random. By subjecting a PRNG to statistical tests, one can determine lack of randomness that is not detectable to our minds. Statistical tests are also not subject to biases such as the Gambler’s Fallacy[[1]](#footnote-1), and can evaluate a generator more efficiently than a human could. The results of statistical tests can help the designers and users of PRNGs to determine where the algorithm’s strengths and weaknesses lie.

Some qualities that good PRNGs should have are:

* a mathematically-proven long period length to prevent wrap-around under reasonable circumstances,
* repeatability,
* uniformity, and the
* availability of jump-ahead methods (which allow one to partition an output sequence into substreams and compute them efficiently).

Additionally, speed, memory requirements, and portability are important considerations [6]. This investigation will use these criteria and statistical tests to explore the question: *How does a pseudo-random number generator’s algorithm affect its quality?*

## Applications of Pseudo-Random Number Generators

Pseudo-random number generators have many applications, including cryptography, simulations, statistical sampling, numerical analysis, and recreation. PRNGs are used to create cryptographic keys and unique nonces. Secure communication over the Internet depends on the production of unpredictable numbers in order to conceal data. Random numbers are also crucial to Monte Carlo simulations, which use randomness to produce a variety of possible outcomes. The Monte Carlo method is used in a wide range of fields, especially in problems with significant uncertainty in inputs. By simulating a real event with a computer program, researchers can compute and evaluate millions of possibilities in a relatively short amount of time. Thus, PRNGs are also useful for operations research. For example, one could test how a network responds to a random node failing. In this case, it is also important that the PRNG output be reproducible. This is another reason a hardware RNG would not be suitable.

Random numbers are also used in numerical approximation in the field of numerical analysis, which give approximate but accurate solutions to difficult problems. Finally, PRNGs are also used for games and gambling. Where you cannot use real die or card decks, electronic casino equipment uses hardware RNGs as an initial source of entropy, in conjunction with a PRNG. Their software is tightly regulated by government organizations. In video games, pseudo-random numbers are used for procedural generation, often of textures and 3D models, but also for decisions in gameplay, such as generation of creatures in the Pokemon games [10].

## Commonalities Among All Pseudo-Random Number Generators

Pseudo-random number generators use an algorithm to determine a random-looking sequence from an initial value, called the *seed*. From a given seed, an algorithm (with the same parameters) will produce the same sequence. At a certain point, the sequence will return to a number already produced and repeat itself. The length between repetitions of a value is called the *period*. The theoretical maximum period length is 2^b for a b-bit machine, as that is the maximum number of values that can be represented. Most PRNG have periods significantly less than this. Many algorithms have theoretically-determined maximum period lengths. A longer period length is generally associated with a higher quality PRNG, while a shorter period length is generally less desirable. However, period length is not the only criteria for randomness.

John von Neumann’s pseudo-random number generator based on the squaring of the middle digits of a number is a notoriously weak PRNG [5]. The maximum period length for the Middle Squares method on an n-digit number is 8^n. However, the main weakness of the Middle Squares method lies in its tendency to converge very quickly for many seed values. For example, the Middle Squares method with a seed of 3792 (—> 6241 —> 576 —> 49 —> 2401 —>1600 —> 3600 —> 3600) converges on 3600. Many other seeds converge very quickly and result in a very short period. Other PRNGs do not have quite as short periods and as apparent weaknesses, but none can be described as truly random. Patterns may emerge from the structure of the PRNG algorithms, which can be identified with statistical tests.

Above all, PRNGs must be tested with their application in mind. If they are intended to be used for cryptography, the results must seem unpredictable, and a human attacker should not be able to know previous outputs if some or all of the PRNG’s internal state internal state is revealed. It should pass the next-bit test, where the next value cannot be predicted with reasonable computational power.

Non-cryptographically secure PRNGs used in statistical sampling and Monte Carlo simulations must focus on the appearance of the output as statistically “random”. PRNGs used in gaming and gambling must also be well-distributed and unpredictable. There are a very large number of statistical tests that could be run on an output sequence, only some of which will be used in this investigation. The vast amount of tests raises the following question: How many statistical tests must be passed by a “good” generator? The general rule is that poor RNGs are those that fail simple tests. Testers must determine which statistical tests carry more weight depending on the RNG’s application. If it is very sensitive to a certain statistical weakness, that test will have greater weight to ensure that the RNG does not compromise the accuracy of their application’s results.

Depending on the context in which they are used, randomness quality may be compromised for practicality. For example, in an action game that depends on the generation of a large sequence of random values very quickly, a less statistically-random but faster PRNG may be used. A cryptographically-secure PRNG that takes a long amount of time for generation may not be chosen for an operations research simulation which does not require security and would be slowed down. Knowing the strengths and weaknesses of common PRNGs can help developers make more informed decisions.

Several common PRNGs will be examined in this paper. The main focus will be on the Linear Congruential Generator (LCG) and the Mersenne Twister (MT). These PRNGs will also be briefly compared to the Secure Hash Algorithm (SHA-1) and Multiple Recursive Generator (MRG).

## Pseudo-Random Number Generators of Interest

#### Linear Congruential Generator

The Linear Congruential Generator (LCG) is a PRNG based on a discontinued piecewise linear equation. It is relatively simple, but the parameters can be modified to reach maximum period length or to make either a multiplicative congruential generator or mixed congruential generator. Java’s default random function (java.util.random) uses an LCG.

An LCG is defined by the equation xi+1 = (axi + c) mod m, where xi  is the previous term, starting from the seed. a is the multiplier, m the modulus, and c the increment. If c = 0, this LCG is a multiplicative congruential generator. If not, it is called a mixed congruential generator. The parameters m, a, and c are generally chosen by the implementor to allow a large period, but can also be chosen to optimize for speed. The period length of a mixed congruential generator (c ≠ 0) is m if and only if the conditions of the Hull-Dobell Theorem are fulfilled [5]:

1. m and c are relatively prime
2. a-1 is divisible by all prime factors of m
3. a-1 is divisible by 4, if m is divisible by 4

For a multiplicative linear congruential generator, the maximum period length m is achieved if and only if a is a primitive root of m (an mod m ≠ 1 for n = 1, 2, ... , m-2) [3]. LCGs run more quickly when their parameters are powers of 2 and when c = 0, as the operations are simpler [5].

#### Mersenne Twister

The Mersenne Twister is another PRNG, whose name derives from the fact that the period length is chosen to be a Mersenne prime, which is one less than a power of two. The most common version of the algorithm is based on the Mersenne prime 219937−1 and is called “MT19937”. The Mersenne Twister developed out of the Generalized Feedback Shift Register (GFSR). There are two parts to Twisted GFSRs: recurrence and tempering.

For a 32-bit implementation, recurrence involves working through an array of 624 32-bit values. From [2], the recurrence relationship is given by xk + n = xk+m ⊕ ( xuk | xlk + 1 ) A . The r leftmost bits of xk (xuk ) are concatenated to the r rightmost bits of xk+1 (xlk+1), then the resulting vector is multiplied by the matrix A. This multiplication can be easily done with simple bit shift operations x>>1 if x0 = 0 or (x>>1) ⊕ a if x0 = 1, where a is the bottom row of matrix A. The bitwise addition operation is performed to add xk+m and ( xuk | xlk + 1 ) A to generate xk + n. This process avoids multiplication and division, increasing speed.

Tempering is performed to increase the k-dimensional equidistribution. Each generated 32-bit word is multiplied by the tempering matrix T, which is chosen such that these bitwise operations are possible, where b and c are tempering bitmasks:

*y* ← *x*[*i*]

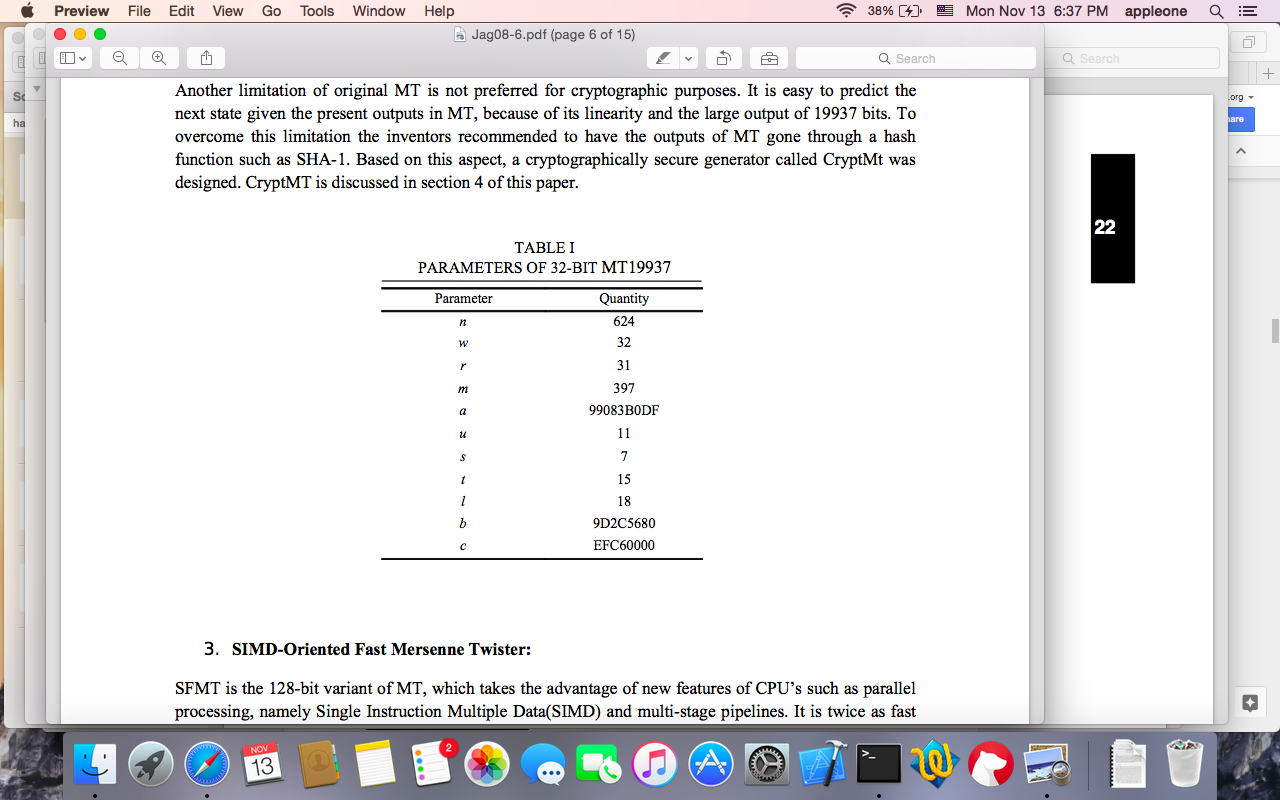
*y* ← *y* ⊕ ( *y* >> *u* )

*y* ← *y* ⊕ (( *y* << *s* ) & *b*)

*y* ← *y* ⊕ (( *y* << *t* ) & *c*)

*z* ← *y* ⊕ ( *y* >> *l* )

The parameters for the MT19937 are given by Table 1 [2]:



Parameters n and r are selected so that nw-r, where w is the word size which in this case is 32 bits, is equal to the Mersenne exponent of 19937. These parameters allow for a long period of 219937-1 and a 623-dimensional distribution. Variations on the Mersenne Twister, such as the Single Instruction Multiple Data (SIMD), attempt to increase its efficiency [2].

The Mersenne Twister is used by the programming languages and software systems Python, R, Ruby, Stata, Mathematica, MATLAB, and many more.

## Summary of Statistical Tests

The PRNGs will be subjected to a battery of statistical tests implemented in the TestU01 library in order to accept or reject the null hypothesis that the numbers are random.

The Crush battery in TestU01 contains 32 different statistical tests, intended to give a stringent evaluation of RNGs. Each test is applied several times with varying parameters to render the tests more thorough. These tests are described in further detail in the TestU01 User Guide [8].

One such test is the Random Walk Test, which computes statistics based on the “random walk” of the random bits over the set of integers Z [11]. It is executed by the function void swalk\_RandomWalk1 (unif01\_Gen \*gen, swalk\_Res \*res, long N, long n, int r, int s, long L0, long L1). It computes test statistics for H, the number of steps the right; M, the maximum value reached; J, the fraction of time spent to the right of the origin; Py, the first passage time at y; R, number of times the walk returned to zero; and C, the number of sign changes.

In order do this, it partial sums of subsequences of length l are computed, where l is a subsequently larger even integer in the interval [L0, L1]. For each bit bi, if bi = 1, then Xi = 1, or if bi = 0, then Xi = -1. The partial sum is given by . The process {, k ≥ 0} is a random walk over the set of integers. For each l even, the test statistics are defined as:

H = steps to the right = =

M = max{, 0 ≤ k ≤ l}

J =

Py = min{k: = y}, y > 0 R =

C =

I[...] denotes the indicator function, which returns 1 if the conditions inside the brackets are true and 0 if not.

Worked-out example:

Sequence = 10000001101110111110

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| k | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|  | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -4 | -3 | -4 | -3 | -2 | -1 | -2 | -1 | 0 | 1 | 2 | 3 | 2 |

l = 20

H = 11

M = 3

J = 6

P, for y = -4: P-4 = 6

R = 2

C = 7

The test generates n random walks, and therefore n values of each statistic. Then, the empirical distribution of these n values with the corresponding theoretical law, given in [8], via a chi-squared test.

The Linear Complexity test is another notable test, as it is consistently failed by pure shift-register type PRNGs [3]. In TestU01, it is composed of two related but separate tests that examine the change in the linear complexity of a sequence of bits. A jump in linear complexity occurs when adding the next bit to the sequence increases its linear complexity, which in the TestU01 software is computed with the Berlekamp-Massey algorithm. The jump complexity test counts the number of jumps. Under the null hypothesis of randomness, for a sample size sufficiently large, the number of jumps is normally distributed (mean and variance formulas given in TestU01 User Guide). The jump size test instead looks at the size of the jumps and counts how many jumps of each size have occurred.

The Lempel-Ziv Compression test determines the compressibility of a string by counting the number W of distinct patterns. The Lempel-Ziv compression algorithm is commonly used as a lossless compression algorithm for GIF images. In this case, it is used on sequences to determine their randomness. Oftentimes, the less random a string is, the more compressible it is (for example, 01010101 is very compressible), but a random string should still be somewhat compressible. The test statistic Z is computed from the number W, and should approximate a standard normal distribution [8].

TestU01’s Crush Battery runs many more tests based on the works of Knuth, L’Ecuyer, Marsaglia, National Institute of Science and Technology (NIST), and many more. It includes the Run, Gap, Coupon Collector, and Simple Poker tests of Knuth [5], the GDC, Birthday Spacings, and Matrix Rank tests of Marsaglia, creator of the Diehard test suite. Some other tests are the Close Pairs, Sample Correlation, Fourier, Hamming Weight, and Autocorrelation tests.

Most of these tests use the chi-squared test to assess “goodness of fit” to reference distributions. Chi-squared distribution is the distribution of X = , where Z1, …, Zk are independent random variables, each of which has a standard normal distribution. A χ2 distribution is said to have k “degrees of freedom”.

Ultimately, what is desired from a statistical test is to determine if the PRNG output is consistent with random behavior. In order to either reject or accept the null hypothesis, the χ2 value of the test statistic is computed, and then compared to the critical value of a chi-squared with a certain degree of freedom. From this, a p-value is generated. In TestU01, if it falls outside the interval [0.001, 0.999], this test is failed and the null hypothesis can be rejected. As seen from the investigation, the results are not that sensitive to the choice of acceptance interval, since the p-values for failing tests tend to be extremely close to 0 or 1, such as 10^-300.

## Investigation

TestU01’s Crush battery uses approximately 34 billion random numbers. It performs 96 different tests on each generator, a given algorithm with specific set of parameters.

#### Comparison of Linear Congruential Generators

Linear Congruential Generators follow the algorithm xi+1 = (axi + c) mod m. If c ≠ 0, it is a Mixed LCG, otherwise it is a Multiplicative LCG. The table below contains a summary of the results of the battery of statistical tests applied to two Multiplicative LCGs and two Mixed LCGs. The code in Appendix A calls functions in TestU01 to create generators with specific parameters and apply statistical tests to their output. The raw results can be found in Appendix D.

Results of Crush battery of tests applied to four different Linear Congruential Generators:

|  |  |  |  |
| --- | --- | --- | --- |
| Multiplicative |  | Mixed |  |
| Apple CarbonLib, C++11's minstd\_rand0 | RANDU | Java.util.Random, POSIX rand48, glibc rand48 | Borland C/C++ |
| m = 2^31-1,  a = 16807,  c = 0,  seed = 12345 | m = 2^31,  a = 65539,  c = 0,  seed = 12345 | m = 2^48,  a = 25214903917,  c = 11,  seed = 12345 | m = 2^32,  a = 22695477,  c = 1,  seed = 12345 |
| 52 tests failed | 131 tests failed | 24 tests failed | 110 tests failed |

The results of a battery of statistical tests applied to various Linear Congruential Generators demonstrates how the choice of parameters greatly influences the output of the PRNG algorithm. The parameters used by RANDU and Borland C/C++ created generators that performed poorly in the statistical tests, while the CarbonLib and Java.util.Random parameters were significantly better.

#### Does each LCG’s parameters allow for maximum period length?

Since the LCG used by Apple CarbonLib, and C++11's minstd\_rand0 is a Multiplicative Linear Congruential Generator with m ≠ 2k, the maximum period length of m-1 is achieved if a is a primitive root of m. Since 16807 is a primitive root of 231 - 1 = 2147483647, the parameters allow it to have a period of 231 - 2. The maximum possible period for a Multiplicative LCG with m = 2k  is 2k-2, achieved if a = 8i± 3 (where i is a positive integer) and the seed is an odd integer. For the RANDU parameters, a = 65539 = 8(8192) + 3, so the maximum period length is 231-2 = 229.

The other two LCGs tested are Mixed Linear Congruential Generators, and must satisfy the Hull-Dobell Theorem for the period to be the maximum length m.

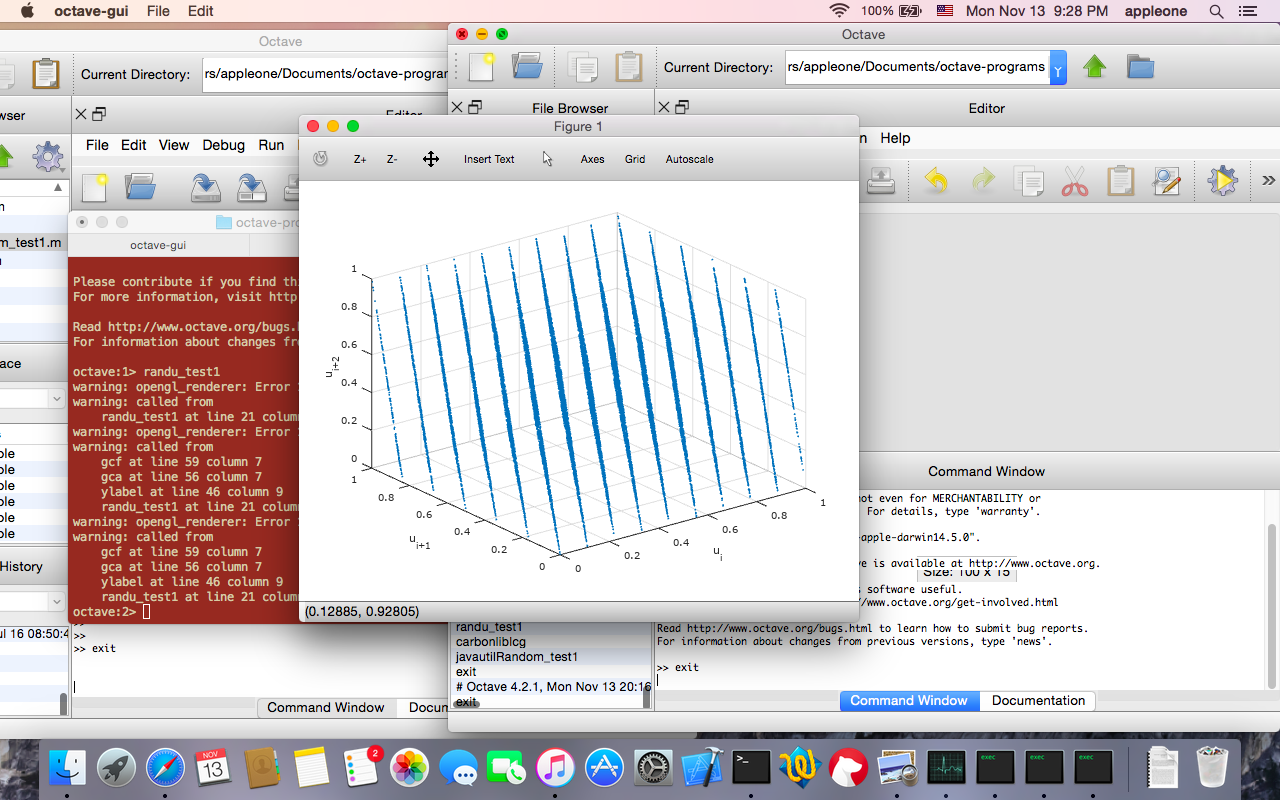
1. m and c are relatively prime
2. a-1 is divisible by all prime factors of m
3. a-1 is divisible by 4, if m is divisible by 4

For the Java.util.Random / POSIX rand48 / glibc rand48 LCG, condition 1 is met: 248 and 25214903917 = 7 \* 443 \* 739 \* 11003 are relatively prime. Condition 2: a - 1 = 25214903916 is divisible by 2. Condition 3: m is divisible by 4, and 25214903916 is divisible by 4.

For the Borland C/C++ LCG, condition 1 is met: 232 and 22695477 = 3 × 72 × 61 × 2,531 are relatively prime. Condition 2: a - 1 = 22695476 is divisible by 2. Condition 2: m is divisible by 4, and 22695476 is divisible by 4.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Generator | Apple CarbonLib, C++11's minstd\_rand0 | RANDU | Java.util.Random, POSIX rand48, glibc rand48 | Borland C/C++ |
| Period length | 231 - 2 | 229 | 248 | 232 |

This difference in period length accounts for some of the discrepancy in performance between these LCG parameters. Naturally, if one PRNG’s period length is 219 times longer than another’s, as is the case with the Java.util.Random LCG versus RANDU’s, the first has more numbers to choose from and is better able to distribute without patterns.



All Linear Congruential Generators present some lattice structure of various dimensions, though many not as obvious as RANDU’s [9]. The poorly-chosen parameters of RANDU led to the undesirable property xk+2 = 6xk+1 − 9xk and a three-dimensional lattice structure, shown in Figure 1.

Figure 1: 3-tuples of 20,000 numbers generated by the RANDU PRNG, plotted by code given in Appendix C

#### Comparison of best LCG, Mersenne Twister, SHA-1, and MRG

The table below contains a summary of the results of the battery of statistical tests applied to 4 PRNGs of different types. The code in Appendix A calls functions in TestU01 to create generators with specific parameters and apply statistical tests to their output. The raw results can be found in Appendix E.

MRG algorithm: xn = (a1xn-1+ … + akxn-k) mod m

Results of Crush battery of tests applied to four different PRNGs:

|  |  |  |  |
| --- | --- | --- | --- |
| Java.util.Random, POSIX rand48, glibc rand48  LCG parameters:  m = 2^48,  a = 25214903917,  c = 11,  seed = 12345 | Mersenne Twister (MT19937) | Secure Hash Algorithm (SHA-1) | Multiple Recursive Generator  Parameters:  m = 2147483647,  k = 2,  A = (46325, 1084587), S = (2, 1) |
| 24 tests failed | 2 tests failed | 1 test failed | 2 tests failed |

It is clear to see from the results of these statistical tests that the Mersenne Twister, SHA-1, and MRG all produce more statistically random values than the Java.util.Random LCG. The Mersenne Twister fails the Linear Complexity test as it is a shift-register type generator [4].

Although Linear Congruential Generators are statistically weaker, they continue to be used because often they are faster, use less memory, and are easier to implement. Very simple operations are performed on one value to generate the next. In many programming languages, LCGs usually only take a few lines to implement - all that is required is a for-loop around a simple equation. The results of speed tests (see below) show that most LCGs run faster than SHA-1 and the MRG, with the exception of the Java.util.Random LCG. This greater CPU time is likely due to the parameter c = 11, the addition of which takes a longer time to perform.

#### Speed Test Results of 108 calls to each generator:

The following table shows the CPU time required to compute 108 random numbers with each generator algorithm. These tests were all run using the C implementations of these algorithms in TestU01. See Appendix B for code the calls to the TestU01 functions to create generators with specific parameters and to invoke the timer function on each generator.

|  |  |
| --- | --- |
| Generator: | Total CPU Time (seconds): |
| Apple CarbonLib, C++11's minstd\_rand0 | 2.15 |
| RANDU | 1.55 |
| Java.util.Random, POSIX rand48, glibc rand48 | 16.23 |
| Borland C/C++ | 2.33 |
| Mersenne Twister (MT19937) | 1.30 |
| SHA-1 | 118.76 |
| MRG | 5.82 |

## 

## Conclusion

The generation algorithm greatly affects the quality of a sequence of pseudo-random numbers and their practicality in various applications. There are several criteria that can help a developer choose the right PRNG for their application.

The first consideration is a PRNG’s performance in statistical tests of randomness, which attempt to identify sequences whose characteristics do not fit theoretical distributions. Many different tests are run in order to expose weaknesses. Failing one test does not mean that a PRNG is necessarily bad. However, if many easy tests are failed, there is a risk that the PRNG will behave non-randomly and negatively affect the application. Of the PRNGs tested, the Mersenne Twister, Secure Hash Algorithm, and Multiple Recursive Generator produce the most statistically random sequences. Of the Linear Congruential Generator (LCG) parameters tested, the parameters used for the Java.util.Random LCG performs the best. However, it fails 24 tests compared to the 1-2 failed by the non-LCG generators tested. While Java.util.Random might be appropriate for a simple game of Solitaire, it would not be suitable for a Monte Carlo simulation or other statistical sampling.

Another factor to consider is speed and memory usage. While the Mersenne Twister, Secure Hash Algorithm, and Multiple Recursive Generator all have minimal statistical weaknesses, the latter two are much slower. Though the Mersenne Twister may slow down with little available RAM, it is usually quite fast due to the use of bitwise operations. SHA-1 makes use of a much larger array, and is therefore quite taxing on time and memory, making it impractical for many applications. The Multiple Recursive Generator is slower than the Mersenne Twister due to its recursion, but faster than SHA-1.

There are also speed differences among the LCGs, particularly between the Java.util.Random parameters and the rest. This is likely due to the c parameter being equal to 11 rather than 0 or 1. This addition takes a significant amount of time to compute. Though the other three perform about eight times faster, Java.util.Random still has its merits in being more “random”.

The last factor to consider is implementation. Depending on the language or platform one wishes to run their application, it may be already implemented. Unless one’s application is especially sensitive to weaknesses in linear complexity, the Mersenne Twister is the strongest choice, as it is very statistically random and efficient. However, if quality is not a priority and there is not already an implementation of the Mersenne Twister in the language, a different PRNG could be chosen. Linear Congruential Generators take very few lines to implement (see Appendix C for a ten-line example), while MT takes upwards of thirty-five lines [2].

Computer scientists have attempted to generate randomness from deterministic machines using many different types of algorithms and hundreds of parameters. Matsumoto and Nishimura’s Mersenne Twister, developed in 1997, appears to be one of the most random-appearing PRNGs, as well as efficient. The Multiple Recursive Generator and SHA-1, which in addition is said to be cryptographically secure, are also statistically strong, but slower. Linear Congruential Generators, though easier to code, are subject to the weakness of a lattice structure that results. Each generator’s algorithm gives it significant strengths and weaknesses that should be kept in mind by PRNG users and developers alike.

## Appendixes

#### Appendix A: C code used to run battery of statistical tests on PRNGS

#include "unif01.h"

#include "bbattery.h"

#include <stdio.h>

#include "ulcg.h"

#include "ugfsr.h"

#include "ucrypto.h"

#include "umrg.h"

int main (void){

unif01\_Gen \*gen;

double x = 0.0;

int i;

// Apple CarbonLib, C++11's minstd\_rand0

gen = ulcg\_CreateLCG(2147483647, 16807, 0, 12345);

bbattery\_Crush (gen);

unif01\_DeleteGen (gen);

// RANDU

gen = ulcg\_CreateLCGFloat (2147483648, 65539, 0, 12345);

bbattery\_Crush (gen);

unif01\_DeleteGen (gen);

// Java.util.Random, POSIX rand48, glibc rand48

gen = ulcg\_CreateLCG (281474976710656, 25214903917, 11, 12345);

bbattery\_Crush (gen);

unif01\_DeleteGen (gen);

// Borland C/C++

gen = ulcg\_CreateLCG(4294967296, 22695477, 1, 12345);

bbattery\_Crush (gen);

unif01\_DeleteGen (gen);

// Mersenne Twister

gen = ugfsr\_CreateMT19937\_98 (1);

bbattery\_Crush (gen);

unif01\_DeleteGen (gen);

//SHA-1

unsigned char mySeed[5] = {0x44, 0x39, 0x78, 0x10, 0x69};

ucrypto\_Mode myMode = ucrypto\_CTR;

gen = ucrypto\_CreateSHA1 (mySeed, 5, myMode, 1, 1);

bbattery\_Crush (gen);

ucrypto\_DeleteSHA1 (gen);

// Multiple Recursive Generator

long A[2] = {46325, 1084587};

long S[2] = {2, 1};

gen = umrg\_CreateMRG(2147483647, 2, A, S);

bbattery\_Crush (gen);

unif01\_DeleteGen (gen);

return 0;

}

#### Appendix B: C code used to run time tests

#include "unif01.h"

#include "ulcg.h"

#include "ugfsr.h"

#include "ucrypto.h"

#include "umrg.h"

#include <stdio.h>

int main (void){

unif01\_Gen \*gen;

double x = 0.0;

int i;

long n = 100000000; // number of values to be generated

//Apple CarbonLibrary/ glibc

gen = ulcg\_CreateLCG(2147483647, 16807, 0, 12345);

unif01\_TimerSumGenWr (gen, n, TRUE);

ulcg\_DeleteGen (gen);

//RANDU

gen = ulcg\_CreateLCGFloat (2147483648, 65539, 0, 12345);

unif01\_TimerSumGenWr (gen, n, TRUE);

ulcg\_DeleteGen (gen);

// Java.util.Random

gen = ulcg\_CreateLCG (281474976710656, 25214903917, 11, 12345);

unif01\_TimerSumGenWr (gen, n, TRUE);

ulcg\_DeleteGen (gen);

//Borland

gen = ulcg\_CreateLCG(4294967296, 22695477, 1, 12345);

unif01\_TimerSumGenWr (gen, n, TRUE);

ulcg\_DeleteGen (gen);

//Mersenne Twister

gen = ugfsr\_CreateMT19937\_98 (123451);

unif01\_TimerSumGenWr (gen, n, TRUE);

unif01\_DeleteGen (gen);

// SHA-1

unsigned char mySeed[5] = {0x44, 0x39, 0x78, 0x10, 0x69};

ucrypto\_Mode myMode = ucrypto\_CTR;

gen = ucrypto\_CreateSHA1 (mySeed, 5, myMode, 1, 1);

unif01\_TimerSumGenWr(gen, n, TRUE);

ucrypto\_DeleteSHA1 (gen);

//MRG

long A[2] = {46325, 1084587};

long S[2] = {2, 1};

gen = umrg\_CreateMRG(2147483647, 2, A, S);

unif01\_TimerSumGenWr (gen, n, TRUE);

unif01\_DeleteGen (gen);

return 0;

}

#### Appendix C: Octave code to produce RANDU three-dimensional plot

% Define parameters

a=65539;

c=0;

x0=1;

m=2^31;

% Calculate sequence using recursion relation

xn=zeros(20000,1);

for i=1:20000

xn(i)=mod(a\*x0+c,m);

x0=xn(i);

end

% Divide by m to give real numbers between 0 and 1

un=xn/m;

% Plot 3-tuples of the u\_i in 3D space

plot3(un(1:end-2),un(2:end-1),un(3:end), ".");

xlabel("u\_i");

ylabel("u\_{i+1}");

zlabel("u\_{i+2}");

grid("on");

#### Appendix D: Results of Statistical Tests on Linear Congruential Generators

|  |  |  |  |
| --- | --- | --- | --- |
| Multiplicative |  | Mixed |  |
| Apple CarbonLib, C++11's minstd\_rand0 | RANDU | Java.util.Random, POSIX rand48, glibc rand48 | Borland C/C++ |
| m = 2^31-1,  a = 16807,  c = 0,  seed = 12345 | m = 2^31,  a = 65539,  c = 0,  seed = 12345 | m = 2^48,  a = 25214903917, c = 11,  seed = 12345 | m = 2^32,  a = 22695477,  c = 1,  seed = 12345 |
| Version: TestU01 1.2.3  Generator: ulcg\_CreateLCG  Number of statistics: 140  Total CPU time: 00:58:52.46  The following tests gave p-values outside [0.001, 0.9990]:  (eps means a value < 1.0e-300):  (eps1 means a value < 1.0e-15):  Test p-value  ----------------------------------------------  1 SerialOver, t = 2 1 - eps1  2 SerialOver, t = 4 1 - eps1  3 CollisionOver, t = 2 1 - eps1  4 CollisionOver, t = 2 1 - eps1  5 CollisionOver, t = 4 1 - eps1  6 CollisionOver, t = 4 1 - eps1  7 CollisionOver, t = 8 eps  8 CollisionOver, t = 8 eps  9 CollisionOver, t = 20 0.9998  10 CollisionOver, t = 20 1 - 2.3e-9  11 BirthdaySpacings, t = 2 eps  12 BirthdaySpacings, t = 3 eps  13 BirthdaySpacings, t = 4 eps  14 BirthdaySpacings, t = 7 eps  15 BirthdaySpacings, t = 7 eps  16 BirthdaySpacings, t = 8 eps  17 BirthdaySpacings, t = 8 eps  18 ClosePairs NP, t = 2 3.2e-157  18 ClosePairs mNP, t = 2 3.2e-157  18 ClosePairs mNP1, t = 2 eps  18 ClosePairs mNP2, t = 2 eps  18 ClosePairs NJumps, t = 2 1 - eps1  19 ClosePairs NP, t = 3 3.2e-157  19 ClosePairs mNP, t = 3 3.2e-157  19 ClosePairs mNP1, t = 3 eps  19 ClosePairs NJumps, t = 3 1 - eps1  20 ClosePairs NP, t = 7 1.8e-79  20 ClosePairs mNP, t = 7 1.8e-79  20 ClosePairs mNP1, t = 7 eps  20 ClosePairs NJumps, t = 7 1 - eps1  21 ClosePairsBitMatch, t = 2 1 - eps1  22 ClosePairsBitMatch, t = 4 1 - eps1  37 Permutation, r = 0 1 - eps1  38 Permutation, r = 15 1 - eps1  39 CollisionPermut, r = 0 2.3e-8  40 CollisionPermut, r = 15 5.0e-11  41 MaxOft, t = 5 eps  42 MaxOft, t = 10 eps  43 MaxOft, t = 20 eps  44 MaxOft, t = 30 eps  50 AppearanceSpacings, r = 20 1 - 1.9e-8  63 GCD, r = 0 2.9e-5  74 Fourier3, r = 0 9.5e-25  75 Fourier3, r = 20 eps  78 PeriodsInStrings, r = 0 1 - 2.3e-7  79 PeriodsInStrings, r = 15 0.9994  80 HammingWeight2, r = 0 1 - 1.6e-6  85 HammingIndep, L = 30 eps  91 Run of bits, r = 0 1 - 1.3e-6  92 Run of bits, r = 20 eps  95 AutoCor, d = 30 1 - 3.3e-10  96 AutoCor, d = 10 1 - 6.7e-6  ----------------------------------------------  All other tests were passed | Version: TestU01 1.2.3  Generator: ulcg\_CreateLCGFloat  Number of statistics: 144  Total CPU time: 00:40:19.30  The following tests gave p-values outside [0.001, 0.9990]:  (eps means a value < 1.0e-300):  (eps1 means a value < 1.0e-15):  Test p-value  ----------------------------------------------  1 SerialOver, t = 2 1 - eps1  2 SerialOver, t = 4 eps  3 CollisionOver, t = 2 1 - eps1  4 CollisionOver, t = 2 eps  5 CollisionOver, t = 4 eps  6 CollisionOver, t = 4 eps  7 CollisionOver, t = 8 eps  8 CollisionOver, t = 8 eps  9 CollisionOver, t = 20 1 - eps1  10 CollisionOver, t = 20 eps  11 BirthdaySpacings, t = 2 eps  12 BirthdaySpacings, t = 3 eps  13 BirthdaySpacings, t = 4 eps  14 BirthdaySpacings, t = 7 eps  15 BirthdaySpacings, t = 7 eps  16 BirthdaySpacings, t = 8 eps  17 BirthdaySpacings, t = 8 eps  18 ClosePairs NP, t = 2 3.2e-157  18 ClosePairs mNP, t = 2 3.2e-157  18 ClosePairs mNP1, t = 2 eps  18 ClosePairs mNP2, t = 2 eps  18 ClosePairs NJumps, t = 2 1 - eps1  19 ClosePairs NP, t = 3 3.8e-28  19 ClosePairs mNP, t = 3 3.2e-157  19 ClosePairs mNP1, t = 3 eps  19 ClosePairs mNP2, t = 3 eps  19 ClosePairs NJumps, t = 3 eps  19 ClosePairs mNP2S, t = 3 eps  20 ClosePairs NP, t = 7 7.0e-8  20 ClosePairs mNP, t = 7 1.8e-79  20 ClosePairs mNP1, t = 7 eps  20 ClosePairs mNP2, t = 7 eps  20 ClosePairs NJumps, t = 7 eps  20 ClosePairs mNP2S, t = 7 eps  21 ClosePairsBitMatch, t = 2 1 - eps1  23 SimpPoker, d = 16 eps  24 SimpPoker, d = 16 eps  25 SimpPoker, d = 64 7.8e-8  26 SimpPoker, d = 64 eps  27 CouponCollector, d = 4 eps  28 CouponCollector, d = 4 eps  29 CouponCollector, d = 16 1.3e-4  30 CouponCollector, d = 16 eps  31 Gap, r = 0 eps  32 Gap, r = 27 eps  33 Gap, r = 0 eps  34 Gap, r = 22 eps  35 Run of U01, r = 0 eps  36 Run of U01, r = 15 eps  37 Permutation, r = 0 eps  38 Permutation, r = 15 eps  39 CollisionPermut, r = 0 2.5e-25  40 CollisionPermut, r = 15 eps  41 MaxOft, t = 5 eps  41 MaxOft AD, t = 5 2.8e-45  42 MaxOft, t = 10 eps  42 MaxOft AD, t = 10 9.7e-40  43 MaxOft, t = 20 eps  43 MaxOft AD, t = 20 1 - eps1  44 MaxOft, t = 30 eps  44 MaxOft AD, t = 30 1 - eps1  45 SampleProd, t = 10 1 - 8.2e-5  46 SampleProd, t = 30 0.9997  49 AppearanceSpacings, r = 0 eps  50 AppearanceSpacings, r = 20 1 - eps1  52 WeightDistrib, r = 8 eps  53 WeightDistrib, r = 16 eps  54 WeightDistrib, r = 24 eps  55 SumCollector 2.6e-8  56 MatrixRank, 60 x 60 eps  57 MatrixRank, 60 x 60 eps  58 MatrixRank, 300 x 300 eps  59 MatrixRank, 300 x 300 eps  60 MatrixRank, 1200 x 1200 eps  61 MatrixRank, 1200 x 1200 eps  63 GCD, r = 0 eps  64 GCD, r = 10 eps  65 RandomWalk1 H (L = 90) eps  65 RandomWalk1 M (L = 90) eps  65 RandomWalk1 J (L = 90) eps  65 RandomWalk1 R (L = 90) eps  65 RandomWalk1 C (L = 90) eps  66 RandomWalk1 H (L = 90) eps  66 RandomWalk1 M (L = 90) eps  66 RandomWalk1 J (L = 90) eps  66 RandomWalk1 R (L = 90) eps  66 RandomWalk1 C (L = 90) eps  67 RandomWalk1 H (L = 1000) eps  67 RandomWalk1 M (L = 1000) eps  67 RandomWalk1 J (L = 1000) eps  67 RandomWalk1 R (L = 1000) eps  67 RandomWalk1 C (L = 1000) eps  68 RandomWalk1 H (L = 1000) eps  68 RandomWalk1 M (L = 1000) eps  68 RandomWalk1 J (L = 1000) eps  68 RandomWalk1 R (L = 1000) eps  68 RandomWalk1 C (L = 1000) eps  69 RandomWalk1 H (L = 10000) eps  69 RandomWalk1 M (L = 10000) eps  69 RandomWalk1 J (L = 10000) eps  69 RandomWalk1 R (L = 10000) eps  69 RandomWalk1 C (L = 10000) eps  70 RandomWalk1 H (L = 10000) eps  70 RandomWalk1 M (L = 10000) eps  70 RandomWalk1 J (L = 10000) eps  70 RandomWalk1 R (L = 10000) eps  70 RandomWalk1 C (L = 10000) eps  72 LinearComp, r = 29 1 - eps1  73 LempelZiv 1 - eps1  74 Fourier3, r = 0 eps  75 Fourier3, r = 20 eps  76 LongestHeadRun, r = 0 eps  77 LongestHeadRun, r = 20 eps  77 LongestHeadRun, r = 20 1 - eps1  78 PeriodsInStrings, r = 0 1 - eps1  79 PeriodsInStrings, r = 15 1 - eps1  80 HammingWeight2, r = 0 eps  81 HammingWeight2, r = 20 eps  82 HammingCorr, L = 30 eps  83 HammingCorr, L = 300 eps  84 HammingCorr, L = 1200 eps  85 HammingIndep, L = 30 eps  86 HammingIndep, L = 30 eps  87 HammingIndep, L = 300 eps  88 HammingIndep, L = 300 eps  89 HammingIndep, L = 1200 eps  90 HammingIndep, L = 1200 eps  91 Run of bits, r = 0 eps  92 Run of bits, r = 20 eps  95 AutoCor, d = 30 1 - eps1  96 AutoCor, d = 10 1 - eps1  ----------------------------------------------  All other tests were passed | Version: TestU01 1.2.3  Generator: ulcg\_CreateLCG  Number of statistics: 144  Total CPU time: 02:14:26.50  The following tests gave p-values outside [0.001, 0.9990]:  (eps means a value < 1.0e-300):  (eps1 means a value < 1.0e-15):  Test p-value  ----------------------------------------------  4 CollisionOver, t = 2 1 - eps1  6 CollisionOver, t = 4 1 - eps1  8 CollisionOver, t = 8 eps  10 CollisionOver, t = 20 eps  11 BirthdaySpacings, t = 2 eps  12 BirthdaySpacings, t = 3 eps  13 BirthdaySpacings, t = 4 eps  14 BirthdaySpacings, t = 7 eps  15 BirthdaySpacings, t = 7 eps  16 BirthdaySpacings, t = 8 eps  17 BirthdaySpacings, t = 8 eps  19 ClosePairs mNP2S, t = 3 4.0e-18  20 ClosePairs mNP2S, t = 7 4.5e-8  24 SimpPoker, d = 16 eps  26 SimpPoker, d = 64 eps  28 CouponCollector, d = 4 eps  30 CouponCollector, d = 16 eps  32 Gap, r = 27 eps  34 Gap, r = 22 eps  38 Permutation, r = 15 eps  40 CollisionPermut, r = 15 1.0e-5  50 AppearanceSpacings, r = 20 3.3e-12  54 WeightDistrib, r = 24 eps  75 Fourier3, r = 20 5.3e-19  ----------------------------------------------  All other tests were passed | Version: TestU01 1.2.3  Generator: ulcg\_CreateLCG  Number of statistics: 140  Total CPU time: 00:54:47.82  The following tests gave p-values outside [0.001, 0.9990]:  (eps means a value < 1.0e-300):  (eps1 means a value < 1.0e-15):  Test p-value  ----------------------------------------------  1 SerialOver, t = 2 1 - eps1  2 SerialOver, t = 4 1 - eps1  3 CollisionOver, t = 2 1 - eps1  4 CollisionOver, t = 2 eps  5 CollisionOver, t = 4 1 - eps1  6 CollisionOver, t = 4 eps  7 CollisionOver, t = 8 1 - eps1  8 CollisionOver, t = 8 eps  9 CollisionOver, t = 20 3.3e-243  10 CollisionOver, t = 20 eps  11 BirthdaySpacings, t = 2 eps  12 BirthdaySpacings, t = 3 eps  13 BirthdaySpacings, t = 4 eps  14 BirthdaySpacings, t = 7 eps  15 BirthdaySpacings, t = 7 eps  16 BirthdaySpacings, t = 8 eps  17 BirthdaySpacings, t = 8 eps  18 ClosePairs NP, t = 2 3.2e-157  18 ClosePairs mNP, t = 2 3.2e-157  18 ClosePairs mNP1, t = 2 eps  18 ClosePairs mNP2, t = 2 eps  18 ClosePairs NJumps, t = 2 1 - eps1  19 ClosePairs NP, t = 3 3.2e-157  19 ClosePairs mNP, t = 3 3.2e-157  19 ClosePairs mNP1, t = 3 eps  19 ClosePairs NJumps, t = 3 1 - eps1  20 ClosePairs NP, t = 7 1.8e-79  20 ClosePairs mNP, t = 7 1.8e-79  20 ClosePairs mNP1, t = 7 eps  20 ClosePairs NJumps, t = 7 1 - eps1  21 ClosePairsBitMatch, t = 2 1 - eps1  22 ClosePairsBitMatch, t = 4 1 - eps1  24 SimpPoker, d = 16 eps  26 SimpPoker, d = 64 eps  28 CouponCollector, d = 4 eps  30 CouponCollector, d = 16 eps  32 Gap, r = 27 eps  34 Gap, r = 22 eps  36 Run of U01, r = 15 eps  37 Permutation, r = 0 1 - 3.2e-13  38 Permutation, r = 15 eps  39 CollisionPermut, r = 0 1 - 2.3e-8  40 CollisionPermut, r = 15 eps  41 MaxOft, t = 5 1 - 8.2e-9  42 MaxOft, t = 10 1 - 3.8e-14  43 MaxOft, t = 20 1 - eps1  44 MaxOft, t = 30 1 - eps1  49 AppearanceSpacings, r = 0 eps  50 AppearanceSpacings, r = 20 1 - eps1  52 WeightDistrib, r = 8 eps  53 WeightDistrib, r = 16 eps  54 WeightDistrib, r = 24 eps  56 MatrixRank, 60 x 60 eps  57 MatrixRank, 60 x 60 eps  58 MatrixRank, 300 x 300 eps  59 MatrixRank, 300 x 300 eps  60 MatrixRank, 1200 x 1200 eps  61 MatrixRank, 1200 x 1200 eps  63 GCD, r = 0 eps  64 GCD, r = 10 eps  65 RandomWalk1 H (L = 90) eps  65 RandomWalk1 M (L = 90) eps  65 RandomWalk1 J (L = 90) eps  65 RandomWalk1 R (L = 90) eps  65 RandomWalk1 C (L = 90) eps  66 RandomWalk1 H (L = 90) eps  66 RandomWalk1 M (L = 90) eps  66 RandomWalk1 J (L = 90) eps  66 RandomWalk1 R (L = 90) eps  66 RandomWalk1 C (L = 90) eps  67 RandomWalk1 H (L = 1000) eps  67 RandomWalk1 M (L = 1000) eps  67 RandomWalk1 J (L = 1000) eps  67 RandomWalk1 R (L = 1000) eps  67 RandomWalk1 C (L = 1000) eps  68 RandomWalk1 H (L = 1000) eps  68 RandomWalk1 M (L = 1000) eps  68 RandomWalk1 J (L = 1000) eps  68 RandomWalk1 R (L = 1000) eps  68 RandomWalk1 C (L = 1000) eps  69 RandomWalk1 H (L = 10000) eps  69 RandomWalk1 M (L = 10000) eps  69 RandomWalk1 J (L = 10000) 1.1e-16  69 RandomWalk1 R (L = 10000) eps  69 RandomWalk1 C (L = 10000) eps  70 RandomWalk1 H (L = 10000) eps  70 RandomWalk1 M (L = 10000) eps  70 RandomWalk1 J (L = 10000) eps  70 RandomWalk1 R (L = 10000) eps  70 RandomWalk1 C (L = 10000) eps  72 LinearComp, r = 29 1 - eps1  73 LempelZiv 2.5e-6  74 Fourier3, r = 0 eps  75 Fourier3, r = 20 eps  77 LongestHeadRun, r = 20 eps  77 LongestHeadRun, r = 20 1 - eps1  79 PeriodsInStrings, r = 15 1 - eps1  80 HammingWeight2, r = 0 1 - eps1  81 HammingWeight2, r = 20 1 - eps1  82 HammingCorr, L = 30 eps  83 HammingCorr, L = 300 eps  84 HammingCorr, L = 1200 1 - eps1  85 HammingIndep, L = 30 eps  86 HammingIndep, L = 30 eps  87 HammingIndep, L = 300 eps  88 HammingIndep, L = 300 eps  89 HammingIndep, L = 1200 eps  90 HammingIndep, L = 1200 eps  92 Run of bits, r = 20 eps  95 AutoCor, d = 30 1 - eps1  96 AutoCor, d = 10 1 - eps1  ----------------------------------------------  All other tests were passed |
| 52 tests failed | 131 tests failed | 24 tests failed | 110 tests failed |

#### Appendix E: Results of Statistical Tests on Various PRNGs

|  |  |  |  |
| --- | --- | --- | --- |
| Java.util.Random, POSIX rand48, glibc rand48  LCG parameters: 2^48, 25214903917 , 11, 12345 | Mersenne Twister | Secure Hash Algorithm (SHA-1) | Multiple Recursive Generator  Parameters: m = 2147483647, k = 2, A = (46325, 1084587), S = (2, 1) |
| Version: TestU01 1.2.3  Generator: ulcg\_CreateLCG  Number of statistics: 144  Total CPU time: 02:14:26.50  The following tests gave p-values outside [0.001, 0.9990]:  (eps means a value < 1.0e-300):  (eps1 means a value < 1.0e-15):  Test p-value  ----------------------------------------------  4 CollisionOver, t = 2 1 - eps1  6 CollisionOver, t = 4 1 - eps1  8 CollisionOver, t = 8 eps  10 CollisionOver, t = 20 eps  11 BirthdaySpacings, t = 2 eps  12 BirthdaySpacings, t = 3 eps  13 BirthdaySpacings, t = 4 eps  14 BirthdaySpacings, t = 7 eps  15 BirthdaySpacings, t = 7 eps  16 BirthdaySpacings, t = 8 eps  17 BirthdaySpacings, t = 8 eps  19 ClosePairs mNP2S, t = 3 4.0e-18  20 ClosePairs mNP2S, t = 7 4.5e-8  24 SimpPoker, d = 16 eps  26 SimpPoker, d = 64 eps  28 CouponCollector, d = 4 eps  30 CouponCollector, d = 16 eps  32 Gap, r = 27 eps  34 Gap, r = 22 eps  38 Permutation, r = 15 eps  40 CollisionPermut, r = 15 1.0e-5  50 AppearanceSpacings, r = 20 3.3e-12  54 WeightDistrib, r = 24 eps  75 Fourier3, r = 20 5.3e-19  ----------------------------------------------  All other tests were passed | Version: TestU01 1.2.3  Generator: ugfsr\_CreateMT19937\_98  Number of statistics: 144  Total CPU time: 01:06:23.01  The following tests gave p-values outside [0.001, 0.9990]:  (eps means a value < 1.0e-300):  (eps1 means a value < 1.0e-15):  Test p-value  ----------------------------------------------  71 LinearComp, r = 0 1 - eps1  72 LinearComp, r = 29 1 - eps1  ----------------------------------------------  All other tests were passed | Version: TestU01 1.2.3  Generator: ucrypto\_CreateSHA1  Number of statistics: 144  Total CPU time: 12:48:47.91  The following tests gave p-values outside [0.001, 0.9990]:  (eps means a value < 1.0e-300):  (eps1 means a value < 1.0e-15):  Test p-value  ----------------------------------------------  19 ClosePairs NJumps, t = 3 2.8e-4  ----------------------------------------------  All other tests were passed | Version: TestU01 1.2.3  Generator: umrg\_CreateMRG  Number of statistics: 144  Total CPU time: 01:12:40.68  The following tests gave p-values outside [0.001, 0.9990]:  (eps means a value < 1.0e-300):  (eps1 means a value < 1.0e-15):  Test p-value  ----------------------------------------------  12 BirthdaySpacings, t = 3 3.7e-14  13 BirthdaySpacings, t = 4 3.3e-5  ----------------------------------------------  All other tests were passed |
| 24 tests failed | 2 tests failed | 1 test failed | 2 tests failed |

## Bibliography

[1] Challis, Vivien. *Pseudo-Random Number Generators*. University Of Queensland: Queensland, 2008.<https://www.maths.uq.edu.au/mathsed/phd/viviannotes.pdf>.

[2] Jagannatam, Archana. *Mersenne Twister – A Pseudo Random Number Generator and its Variants*. George Mason University: Fairfax, VA, 2008. <http://ece.gmu.edu/~jkaps/scholarlypaper.html>

[3] Jain, Raj. Presentation on *Random-Number Generation*. Washington University: St.Louis, 2008. <http://www.cs.wustl.edu/~jain/cse567-08/ftp/k_26rng.pdf>

[4] Jones, David. *Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications*. UCL Bioinformatics Group: London, 2010. <http://www0.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf>

[5] Knuth, Donald. *The Art of Computer Programming, Seminumerical Algorithms, 3rd ed.* Stanford University: Addison-Wesley, 1997.

[6] L’Ecuyer, Pierre. *Random number generation.* Department of Information and Operations Research at the University of Montreal: Montreal, 2012. <https://www.iro.umontreal.ca/~lecuyer/myftp/papers/handstat.pdf>

[7] L'Ecuyer, Pierre, and Richard Simard. "Testu01." *ACM Transactions On Mathematical Software*, vol 33, no. 4 (2007): 22-es. *Association For Computing Machinery (ACM)*, doi:10.1145/1268776.1268777.

[8] L'Ecuyer, Pierre, and Richard Simard. *TestU01 User’s guide, compact version.* Department of Information and Operations Research at the University of Montreal: Montreal, 2013. <http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf>

[9] Marsaglia, George. ["Random numbers fall mainly in the planes."](http://www.pnas.org/content/61/1/25.full.pdf) *Proceedings of the National Academy of Sciences*, vol 61, no. 1(1968): 25-28. *National Academy of Sciences.* <http://www.pnas.org/content/61/1/25.full.pdf>

[10] "Pseudorandom Number Generation In Pokémon". 2016. Bulbapedia. [https://bulbapedia.bulbagarden.net/wiki/Pseudorandom\_number\_generation\_in\_Pokémon](https://bulbapedia.bulbagarden.net/wiki/Pseudorandom_number_generation_in_Pok%C3%A9mon).

[11] Rukhin, Andrew et al. *A Statistical Test Suite For Random And Pseudorandom Number Generators For Cryptographic Applications*. National Institute Of Standards And Technology: Gaithersburg, MD, 2010.<http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf>.

1. The Gambler’s Fallacy is the mistaken belief that if one result has been produced frequently in the past, then the other result is due to happen with a chance greater than 50% in the future. [↑](#footnote-ref-1)