Random Number Generators

Chris Dibble and Hannah Turner

Chris Dibble graduated Summa Cum Laude from Ball State University in 2013 with a degree in computer science and a minor in mathematics. He is currently employed as a Software Engineer at Interactive Intelligence in Indianapolis, Indiana. This paper is based on a project he completed in conjunction with Hannah Turner for an honors colloquium "Chaos Theory and Fractals" taught by Dr. Rich Stankewitz.

Hannah Turner graduated this year from Ball State University with a Bachelor's in mathematics. She will study mathematics in the Budapest Semesters in Mathematics during the 2014–2015 academic year supported by a Fulbright grant. When she returns she plans to enroll in a PhD program in mathematics.

Introduction

From simple gambling thousands of years ago, to modern statistical sampling, humans have created random number generators. Methods ranging from a simple coin flip to complex deterministic equations have been employed for amusement, statistics, and science. However, the use of random number generators, even today, is still not completely understood.

Our foray into random number generation barely scrapes the surface of the subject. Thus, much of this work is a compilation of previous research. Our aim has been to condense the wealth of information on random number generation into a simple overview and further, execute several established generators. Through original computer applets, we attempted to test the degree of randomness of the generators, using several traditional statistical tests.

Random Number Generators

As one can imagine, random number generation is quite a broad topic. Dodge, in the Concise Encyclopedia of Statistics offers a definition: "Random number generation involves producing a series of numbers with no recognizable patterns or regularities, that is, appearing random."

Thus, computers do not hold a monopoly on random number generation. Dice rolls, coin flips, drawing lots, and other forms of physical means can indeed produce streams of numbers that one could not predict. These were the earliest random number generators. As larger and larger streams of numbers were required for scientific procedures, physical random number generators lost value; scientists required the streams within a reasonable time-frame, with little effort. With the advent of computing, random number generation received a needed modern boost.

A computer is capable of two distinct types of random number generation. The first is comparable to a physical random number generator. What is known as a hardware random generator can extract information from a "weakly" random physical process, and convert the information into a stream of numbers. Processes used include thermal noise, nuclear decay, or "quantum" processes, such as the agitation of electrons.

Computers also excel at iterating deterministic equations. Generators based on a deterministic, iterated function are called pseudorandom number generators. These generators are the primary ones used in scientific research in our modern world. These three distinct genres of generators offer unique ways of random number generation. In studying these generators, one can learn much about randomness, and its applications in science.

What is Random?

To see the value in random number generation, one needs to know what is meant by a random number. For accuracy, we refer to Edwin Landauer, in his "Methods of Random Number Generation," [5] to define the term. "A random number is defined to be a series of digits where (a) each digit has the same probability of occurring and (b) adjacent digits are completely independent of one another (if we know one digit, we still have no way of predicting the next one)." That is, given any number of digits in a stream of digits, we could not predict the following, or previous numbers, for example.

This definition is, of course, not all-encompassing. Many degrees of randomness in streams of "random" numbers exist. In any deterministic generator, the streams of numbers created cannot be called "random" since they are created through an algorithm. Thus, they are known as "pseudo-random" numbers. Up to the discretion of the tester, a random number generator may be designated "statistically random" if it passes certain chosen tests. For many purposes, "statistical randomness" is acceptable, though such "random" streams are not truly random, and may even fail to "appear" random if further testing is employed.

However, one can estimate the degree of statistical randomness of a generator. There are many varieties of statistical tests, some of which we will discuss in detail in subsequent sections.

Modern Uses of Random Numbers

One of the most famous uses for random numbers are those in statistical sampling. Many facets of statistics require random numbers, such as choosing a representative sample and randomization of steps of an experiment to hide the protocol from the subject. It is also useful in creating the design itself. Certain types of statistical analysis require random number generators. In any of these uses, a faulty random number generator can nullify any results. Simulation of statistical events such as a coin toss, or computer simulation of physical phenomenon also requires random number generators.

Cryptography is another interesting application of random number generation. For the utmost security, the streams of numbers used must not only be random, but also from an unknown, that is, new source. For example, the digits of π can provide random streams. According to Weisstein [11], the decimal digits of π, are thought to be normal, that is, the distribution of its decimal digits $(0-9)$ has a uniform probability of $1/10$ for each digit. There has been as of yet no widely accepted proof that any non-artificially constructed numbers are normal. However, the first $30,000,000$ decimal digits of π have been shown to be uniformly distributed (Bailey, [1]). One might think that drawing from a possibly uniform distribution, like the digits of π, would be an efficient method of generation. However, well known streams like that of π, or commonly used deterministic generators are not very secure options.

Finally, Monte Carlo methods of simulation are yet another remarkable application of random numbers. This is a broad heading under which fall many algorithms which, instead of using a deterministic function, use random sampling to compute results. Weisstein [13] defines a Monte Carlo method as "any method which solves a problem by generating suitable random numbers and observing that fraction of the numbers obeying some property or properties". A common use of the Monte Carlo method is known as Monte Carlo integration. To simplify integration of a function with a complex domain, D, Monte Carlo integration randomly plots "many" points over a larger, simpler domain; each point can then be categorized: it is either inside the bounds of D, or outside. An estimation of area can then be made using the ratio of points in D to those outside of D but within the much simpler domain ([13]).

Before this method came about, simulations were tested against already solved deterministic problems to test their success. In turn, Monte Carlo methods solve deterministic problems using a simulation. They are usually applied to problems with many variables and parameters, that is, a system too complex to solve deterministically. They could therefore be applied to dynamical systems; though not necessarily arising from complex multivariate equations, these systems are often difficult to examine deterministically. Monte Carlo methods have, in fact, been applied to fluid flow, for example.

Physical Methods

Historical Methods

In a sense, the earliest methods of random number generation were highly successful. Physical processes are not entirely random, but are usually so complex that they are unpredictable to the casual observer.

In the ancient world, chance was linked to fate. Those interested in their fate could throw dice in the hopes of divining their future. Primitive "generators" included the scattering of rice and the interpretation of the cracks of a turtle's shell, once heated. These methods were not number generators, as they did not produce numbers; they required interpretation before recording. Actual random number generators were used as well. Specifically, in the I Ching, an ancient Chinese text, there are many methods discussed in depth of random number generation. These include coin-flipping and the counting of yarrow stalks. The random data produced by these methods was then used to divine fate.

These methods of divination morphed into games of chance, the first being simple bets on the rolls of a die. Games of chance date back to 2000 B.C.

Generating Numbers from the Environment

In our modern world, using physical processes to generate random streams may seem outdated. A deterministic random number generator, like those discussed in the next section, can speed one's work along substantially. However, there are instances in which such a generator is not useful. This is true in certain facets of cryptography. In cryptography, random numbers are used to generate keys for secure communication. If the numbers used are not random enough, security may be compromised.

In cryptography and other areas, ultra-secure options include generators called hardware random number generators. These are based on physical processes like the historical generators mentioned. However, the processes used for the hardware generator are favored to be those with quantum quality, that is, quantum mechanics predicts these phenomena are fundamentally random. These include shot noise, which is "noise" created when photons from a lamp are directed to a photodiode. Noise occurs when the photons affect the circuit. Other phenomena are used which are not predicted random by quantum mechanics; these include thermal noise: the agitation of electrons inside a conductor. Quantum processes are hard to detect, as they are so small, thus other sources of noise are used, though these are more easily attacked.

From the physical noise, a transducer converts the physical phenomenon into an electrical signal. Then, an amplifier is usually required to increase the amplitude of the signal to the macroscopic level, so it can be read by an analog to digital converter that finally yields a number.

While generally very hard to predict, there are often issues of statistical randomness in the final data. One example is called bias. The data will often have a mean not equal to the expected 0.5 of a uniform distribution on the interval $[0, 1]$. Thus most hardware generators have what is called a corrector to bring the mean to the desired value. Another problem can arise if data is sampled too quickly. In such a scenario, short-term dependencies of physical phenomenon may cause the data to be correlated. Finally, hardware generators are sensitive to outside noise. This can effectually ruin a data set. Roger Davies, in his Hardware Random Number Generators, presents two spectral analyses of hardware random number generators, one that passes the significance test, and one that is badly contaminated by an outside frequency, and thus fails miserably (See Figure 1 and Figure 2 below). Note the difference between the distribution of the data between the horizontal lines in each graph .

Though these generators are often slow, the field is still expanding. A recent development is the use of lasers as a source of random numbers. Especially exciting in the new study, multiple sources were used, thus multiple streams of data were generated concurrently. Setups like this one could help speed hardware generators considerably in the future (Wu, [9]).

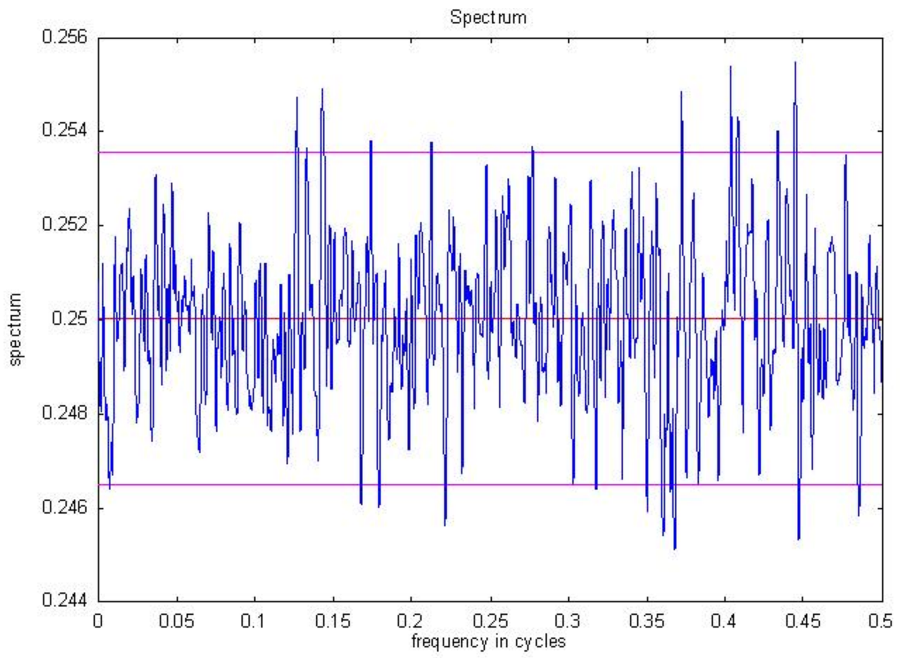


Figure 1

Software Random Number Generators

Casting lots or using environmental white noise to generate random numbers is not always possible, or feasible. In many modern scenarios, casting lots or cracking turtle shells simply takes too much time. In a database where user information is encrypted with random prime numbers, rolling dice to deter-

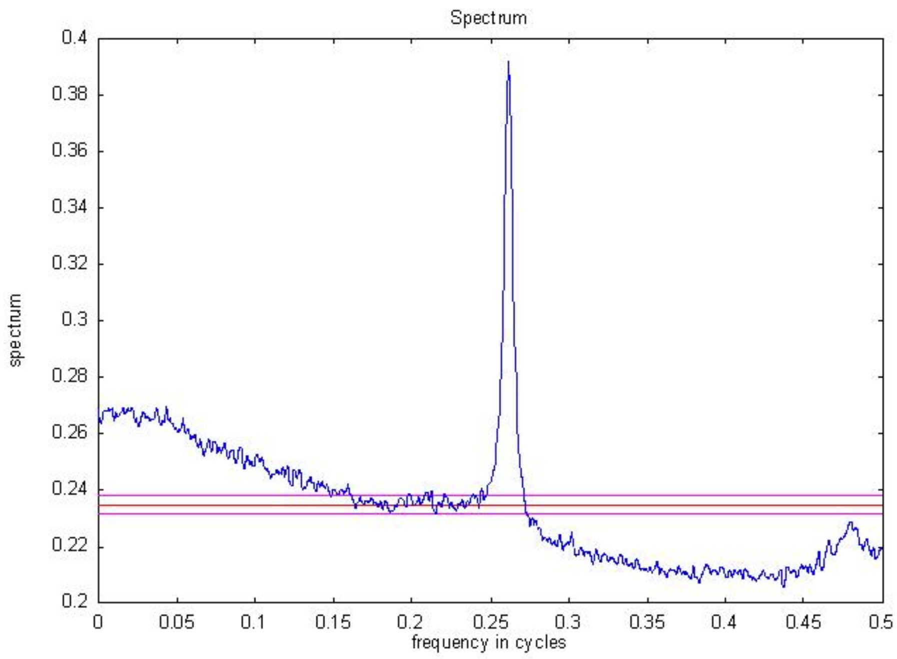


Figure 2

mine numbers is impractical, to say the least, when dealing with thousands of users. Generating the numbers from the environment often takes time as well as dedicated computer hardware. The specialized computer hardware makes these generators impractical for most users. Software random number generators allow for the creation of high quality random number sequences quickly and without any specialized computer hardware.

The algorithms used in software random number generators are easily implemented on computers for use in cell phones, laptops, servers, and data centers. These algorithms have multiple parameters that dictate how the generator will perform. In most cases, these parameters are preset values except for a seed value. The seed value is typically time based and is used as a starting place for the production of the sequence. Unfortunately, these algorithms all have a cyclic period after which they will repeat. There are guidelines available for each generator to maximize its period. Sometimes the restrictions make selecting the proper initialization for a generator a difficult process. However, modern programming languages come with these random number generators built-in so that software engineers always have access to good random number sequences. There are many different algorithms for generating a sequence of pseudorandom numbers. Some of the most commonly used are discussed below.

Linear Congruential Generator

The most common category of software random number generators are the Linear Congruential Generators (LCG). An LCG is defined as the recurrence relation of the form:

$$\begin{aligned}&X_{n+1} = (aX_n + c)(\bmod m),\ n \geq 0\\&m, \text{the modulus};\ m > 0\\&a, \text{the multiplier};\ 0 \leq a < m\\&c, \text{the increment};\ 0 \leq c < m\\&X_0, \text{the seed value};\ 0 \leq X_0 < m.\end{aligned}$$

According to Knuth [3], LCG's are known to have a period of length m if three conditions are met. First, c and m must be relatively prime. Second, a should be divisible by all prime factors of m. Third, a should be a multiple of 4 if m is a multiple of 4. In addition, the choice of m is typically the word size of the computer (usually 2^{32} or 2^{64}) because that makes the binary modular operation very fast. Generator's whose parameters do not meet these requirements have significantly shorter periods and are therefore not suitable for many scenarios. LCG's excel in applications where a small memory footprint is crucial, such as in embedded systems like an internet router or car computer, because they need very little memory in order to execute.

Lehmer Generator

The Lehmer Generator satisfies the relation:

$$\begin{aligned}&X_{n+1} = (a\,X_n)(\bmod m),\ n > 0\\&m, \text{the modulus};\ m > 0\\&a, \text{the multiplier};\ 0 \leq a < m\\&X_0, \text{the seed value};\ 0 \leq X_0 < m\end{aligned}$$

The Lehmer Generator is a special case of the Linear Congruential Generator. Namely, an LCG with $c = 0$. This generator is slightly faster at generating numbers, but that speed increase comes with added restrictions and limitations. The period of a Lehmer Generator cannot be m because c and m are not relatively prime. However, Knuth states that if three conditions are satisfied by m, a, and X_0, a period of $m - 1$ may still be obtained. The conditions are that (1) X_0 be relatively prime to m, (2) a be a primitive element modulo m, and (3) m be prime. For systems where the speed is more important, the Lehmer generator can save some time, albeit at the cost of more restrictions.

Additive Lagged Fibonacci Generator

The Additive Lagged Fibonacci Generator takes on the form:

$$S_n = S_{n-j} + S_{n-k}(\bmod m),\ 0 < j < k,$$

where for most applications m is power of 2, that is $m = 2^M$. This generator is an evolutionary progression of the generator:

$$S_{n+1} = S_n + S_{n-1} (\bmod m)$$

which fails many of the statistical tests for evaluating generators and is used as a stereotypical "bad" example.

The values of j and k must be very carefully selected for the generator to achieve its maximum period of $(2^k - 1)2^{M-1}$. According to Knuth, this period is only achieved if and only if (1) modulo 2, the sequence has period $(2^j - 1)$, (2) modulo 2^2, the sequence has period $(2^j - 1)2$, and (3) modulo 2^3, the sequence has period $(2^j - 1)2^2$.

The benefit of the Fibonacci generator is that it has a very large period when compared with the LCG. It is also very fast if implemented properly. Knuth cites that these generators have performed outstandingly in many empirical tests since their inception in 1958.

The downside to these generators, according to Knuth, is that "there is still very little theory to prove that it does or does not have desirable randomness properties." This may be a large deterrent since the theory behind these generators is largely unknown and therefore carries more inherent risk.

Testing Software Random Number Generators

Random number generators need to be tested to ensure that they are, in fact, sufficiently random. Without robust tests, generators that appear correct may hide significant flaws. There have been many tests for random number generators developed over the years. Two of the most well known are outlined below.

Spectral Test

The spectral test is used to find good values for the multiplier of a given Linear Congruential Generator. It relies on the fact that LCG's will generate numbers that create a lattice structure if they are plotted using successive numbers from the sequence as x-y pairs, i.e., plotting (X_n, X_{n+1}). Figure 3 and Figure 4 show two different examples of this lattice structure using an LCG with $c = 2, m = 256$. The multiplier is set to 61 and 101, respectively.

Plotting the points in higher dimensions also yields this same lattice structure. The spectral test looks at the maximum distance between two of the adjacent, parallel lines, planes, or hyperplanes (depending on dimension). The goal of a good multiplier will be to minimize this distance. From Figures 3 and 4, it's easy to see that $a = 101$ (Figure 4) yields better results than $a = 61$. In this way, it is easy to compare two different values of a with a simple visual test.

There are also ways of determining numerically whether or not a generator passes the spectral test. Section 3.3.4 of Knuth's "The Art of Computer Programming" outlines the procedure for computing a generator's score and

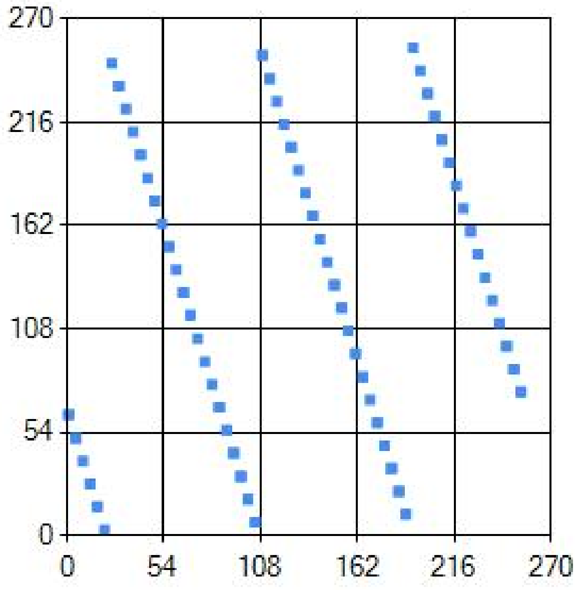


Figure 3

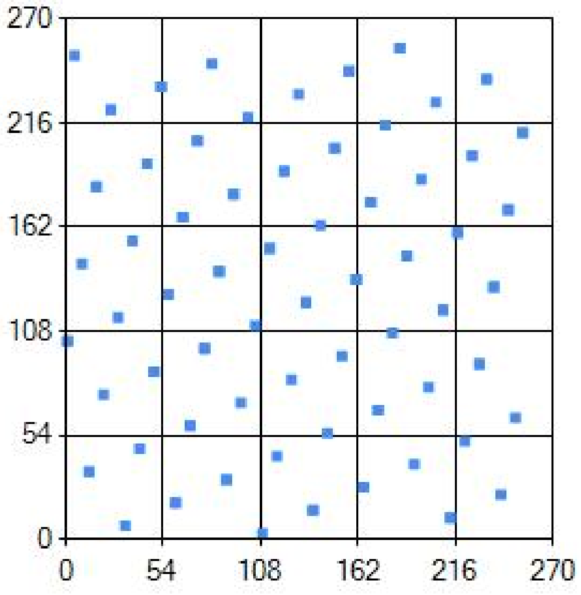


Figure 4

also gives the ranges of acceptable scores for different generator types. While a discussion of the numerical process and its theory is outside the scope of this investigation, a major takeaway from Knuth's text is that the spectral test is the best test for LCG's because it will accurately fail generators that pass other tests. This is possible because the test analyzes the generator's performance for all dimensions 2 through 6. The spectral test was used to show that something called the RANDU generator, which was used for years in research, was not an effective generator because it failed the test for higher dimensions.

Parking Lot Test

The Parking Lot Test is another way of measuring the effectiveness of a generator. It looks at how many disks of diameter 1 a generator can successfully "park" on a 100×100 grid and compares it to a known average number computed by Intel for use in its Math Kernel Library 11, version 2. The test consists of ten subtests, whose results are combined to determine whether or not the test on a whole should pass. Each of the ten subtests consists of using the generator to generate $12,000$ x-y coordinates. These coordinates each represent the center of a disk with diameter 1. After the disk locations are generated, the test will attempt to park the disks at their location. A disk is considered parked if it does not overlap with any other parked disk. If it overlaps, the disk is not placed, and is instead discarded. At the end of the subtest, the number of successfully placed points is used to calculate a p–value for the test. The p–value is calculated as

$$p = \Phi((K - a)/\sigma),$$

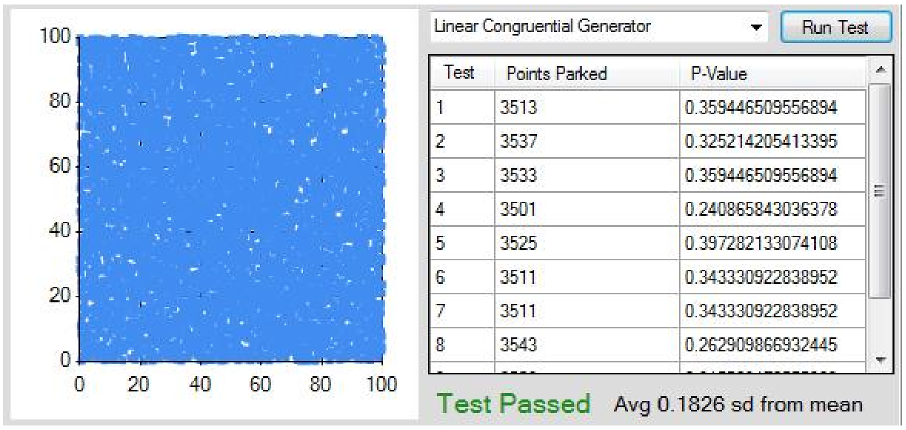
where a is the theoretical mean, σ is the theoretical standard deviation, and K is the observed number of parked disks. It makes use of the phi cumulative distribution function Φ. Numerous experiments conducted by Intel have shown that the theoretical mean and standard deviation should be $3,523$ and 21.9, respectively.

The p–value is recorded for each of the 10 tests. An individual subtest is considered to be passing if its p–value is between 0.05 and 0.95, exclusive. If 5 or more of the subtests pass, the test as a whole passes. Below are graphical representation of the test, generated using a $C\#$ application written by the first author. This application is available freely on the web at
`https://www.dropbox.com/s/znuqsd9u3sckey4/GeneratorApplication.exe`

In Figures 5, 6, a graph shows the plotted disks, while the table on the right shows the number of disks parked as well as the p–values for the subtests. The final result of the test is displayed on the last row of the table.

Conclusions

Random number generation is a veiled subject; often used in mathematical settings, but not much studied. Our foray into number generators yielded an implementation of several software generators that seem to work well given certain seed values and parameters. Other choices of these values failed spectacularly. Given our limited knowledge on the barrage of tests available to test

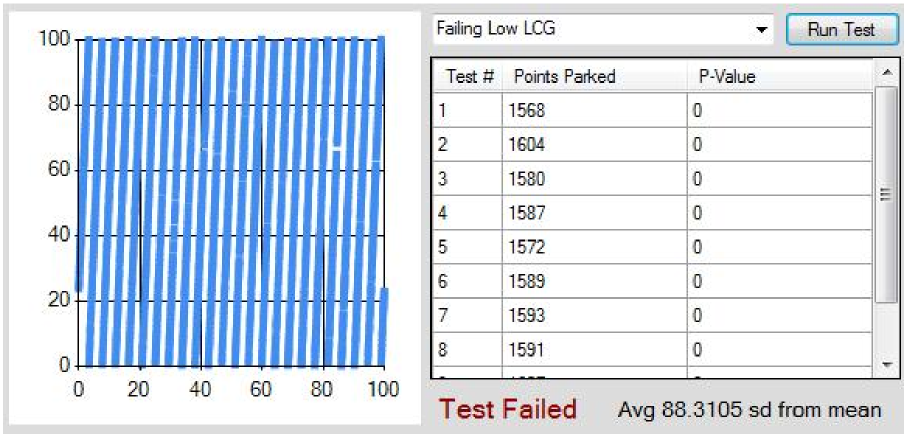


Linear Congruential Generator

Test	Points Parked	P-Value
1	3513	0.359446509556894
2	3537	0.325214205413395
3	3533	0.359446509556894
4	3501	0.240865843036378
5	3525	0.397282133074108
6	3511	0.343330922838952
7	3511	0.343330922838952
8	3543	0.262909866932445

Test Passed Avg 0.1826 sd from mean

Figure 5



Failing Low LCG

Test #	Points Parked	P-Value
1	1568	0
2	1604	0
3	1580	0
4	1587	0
5	1572	0
6	1589	0
7	1593	0
8	1591	0

Test Failed Avg 88.3105 sd from mean

Figure 6

these generators, we cannot say definitively whether any of these generator would continue to perform, under more extensive testing. What we can report is the complexity of random number generation. A deeper understanding of such methods is required as we rely increasingly on random number generation globally.

References

[1] D. H. Bailey, *The Computation of* π *to* $29,360,000$ *Decimal Digit using Borwein's' Quartically Convergent Algorithm*, Math. Comput. **50** (1988) 283–296

[2] Parking Lot Test, Vector Statistical Library Notes for Intel Math Kernel Library 10.3 Update 12, Intel Corperation, n.d. Tue. 27 Nov. 2012. `http://software.intel.com/sites/products/documentation/doclib/mkl_sa/11/vslnotes/8_3_10_Parking_Lot_Test.htm`.

[3] Donald E. Knuth, Generating Uniform Random Numbers, Seminumerical Algorithms, 2nd ed., Reading, MA: Addison-Wesley Pub. **2** (1981) 9–38. Print. The Art of Computer Programming.

[4] Donald E. Knuth, The Spectral Test, Seminumerical Algorithms. 2nd ed., Vol. 2, Reading, MA: Addison-Wesley Pub. **2** (1981) 89–114. Print. The Art of Computer Programming.

[5] Edwin G. Landauer, *Methods of Random Number Generation*, The Two-Year College Mathematics Journal **8** (5)(1977) 296–303. Retrieved from `http://www.jstor.org/stable/3026786?seq=1`

[6] Other Random Number Generators, GNU Scientific Library – Reference Manual, GNU, n.d. Web. 27 Nov. 2012. `http://www.gnu.org/software/gsl/manual/html_node/Other-random-number-generators.html`.

[7] Mascagni, Michael, M. L. Robinson, Daniel V. Pryor, and Steven A. Cuccaro, Parallel Pseudorandom Number Generation Using Additive Lagged-Fibonacci Recursions, Tech. Supercomputing Reaearch Center, I.D.A., n.d. Web. 27 Nov. 2012. `http://www.cs.fsu.edu/~mascagni/papers/RIJP1995_1.pdf`.

[8] Dodge, Yalodah, The concise encyclopedia of statistics, New York, Springer 2008.

[9] J. G. Wu, X. Tang, Z. M. Wu, G. Q Xia, and G. Y Feng, *Parallel generation of* 10 *gbits/s physical random number streams using chaotic semiconductor lasers*, Laser Physics, **22** (10) (2012) 1476–1480. Retrieved from `http://link.springer.com/article/10.1134%2FS1054660X12100246`

[10] R. Davies, Hardware random number generators, 15th Australian Statistics Conference, 2000. Retrieved from `http://www.robertnz.net/hwrng.htm`

[11] Eric W. Weisstein, Normal Number, From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/NormalNumber.html`

[12] Eric. W. Weisstein, Normal Number, From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/Mont`

[13] Eric W. Weisstein, Monte Carlo Integration, From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/MonteCarloIntegration.html`