

Chapter One

Introduction

Randomness and random numbers have traditionally been used for a variety of purposes, for instance, games. With the advent of computers, people recognized the need for a means of introducing randomness into a computer program. Surprising as it may seem, however, it is difficult to get a computer to do something by chance. A computer running a program follows its instructions deterministically and the result is therefore completely predictable.

Computer engineers chose to introduce randomness into computers in the form of pseudo-random number generators. As the name suggests, pseudo-random numbers are not truly random. Rather, they are computed from a mathematical formula or simply taken from a pre-calculated list. A lot of research has devoted to the pseudo-random number theory such that many modern random number generating algorithms are so good that the generated numbers look like they were truly random. However, as expected, pseudo-random numbers have the characteristic of predicability, meaning the whole sequence can be predicted if one knows where in the sequence the first number start.

Pseudo-random numbers are used for computer games but also used on a more serious scale for the generation of cryptographic keys, for simulation, and for some scientific experiments. For cryptographic use, however, it is important that the numbers used to generate keys are not just seemingly random; they must be unpredictable. All strong (meaning difficult to crack) cryptography requires truly random numbers to generate keys. For simulation, there is a whole set of numerical "Monte Carlo" techniques to base on. Simulation studies are commonly used in evaluation of newly developed statistical methods, or when analytical evaluation of quantities of interest is extremely difficult. Therefore, quality of any simulation study depends heavily on the quality and quantity of the uniform random numbers generated. For example, if one conducts a simulation study with a large number of repetitions once the generator runs through its period, the generated random numbers start cycling and no longer "independent." Selection of the seed does not remedy this problem. The seed is the point at which the generator starts in a sequence of period k . Most random number generators have a default

seed of 0 and allow alternative seeds set by the user. A seed of 0 typically invokes an algorithm that calculates the take-off point within the sequence from the system date and time of the computer. Thus, running a program at different times, it will generate different sequences of numbers; but within each sequence, intrinsic periodicity is unavoidable. Thus, having a very long cycle becomes an essential requirement for a good random number generator (RNG) nowadays.

In Chapter 2, we introduce the following pseudo-random number generators. LCG (Linear Congruential Generator) was proposed by Lehmer in 1949 [16]. It is one of the oldest, most studied, and most popular methods for generating random numbers [11]. As the computational power gets cheaper, increasingly long sequences of random numbers are used in modern applications. Reliable generators with longer periods become a necessity. MRG (Multiple Recursive Generator) extends the LCG by higher order recursion to obtain longer period and other good properties. However, MRG has a drawback in computing efficiency: its computing time is about k times slower than that of LCG. Viewing this, Deng and Lin [3] proposed FMRG (Fast MRG) in 2000 by setting as many coefficients α_i in MRG to be 0 or ± 1 as possible to improve the computing efficiency. Continually improving the generators, Deng and Xu [4] recently proposed a system of High-dimensional, Efficient, Long-cycle and Portable (HELP) uniform random number generators named DX random number generators. It is based on some special forms of MRG corresponding to primitive k -th degree polynomials.

The main objective of this study is to investigate the performance of the DX random number generators empirically. We test the DX RNGs by the Diehard test suite, which is a collection of test programs provided by Marsaglia [18]. In Chapter 3, we describe the experimental design of our empirical study. Results are given in Chapter 4. As a result, the DX random number generators passed almost all the tests in Diehard. Pierre L'Ecuyer [13] once said that no random number generator can pass all statistical tests and a proper way to evaluate RNGs is: a bad RNG is one that fails simple tests, whereas a good RNG is one that fails only complicated tests. In this sense, we may claim that the DX RNGs is a system of good random number generators based on the results of our empirical study.

Chapter Two

Literature Review

2.1 Pseudo-random Number Generators

A pseudo-random number generator (PRNG) is a deterministic algorithm that generates a sequence of numbers with little or no discernible pattern in the numbers. Any computer program can only generate pseudo-random numbers. Thus, the outputs of pseudo-random number generators are not truly random. They can only approximate some of the properties of random numbers. Nonetheless, pseudo-random number generation is an important part of modern computing, from cryptography to Monte Carlo methods for simulating physical systems. Careful mathematical analysis is required to ensure that the generated numbers are sufficiently “random”.

Most such algorithms attempt to produce samples that are uniformly distributed. Because a PRNG is a deterministic algorithm, its output has certain properties that a truly random sequence would not exhibit. One of these is the guaranteed periodicity. A generator that is not periodic can be designed, but its memory requirements would slowly grow as it runs [11]. In addition, a PRNG can start from an arbitrary starting point, or seed state, and will always produce an identical sequence from that point on. In practice, many PRNGs exhibit artifacts that may cause them to fail some statistical significance tests. These include: shorter than expected periods for some seed states, poor equi-distribution property, successive values may not be independent, some bits are more random than others, and lack of uniformity.

2.2 Linear Congruential Generators: LCG

An LCG has three integer-valued parameters denoted by B , C , and M , respectively. This class of generators uses a method similar to the folding schemes in chaotic maps. Its basic form is

$$X_i = (BX_{i-1} + C) \bmod M, \quad i \geq 1,$$

where B and C are relatively prime numbers. B is known as the multiplier, C is the

increment, and M is the modulus. When $C=0$, the LCG is called the multiplicative LCG (MLCG).

The choice of $B=16807$, $C=0$, $M=2147483647$ is a very popular set of parameters for the LCG. These parameters were published by Park and Miller [20]. This particular generator, often known as the minimal standard random number generator, is often but not always the generator used for the built-in random number function in compilers and other software packages.

There are two characteristics of LCGs. One is Periodicity and the other one is Parallel Hyperplanes. Given an initial seed x_0 , there is some $n \leq M$ such that $x_n = x_0$. We say the periodicity of this LCG is the least such n . The existence of period can be proved by a simple application of the Pigeonhole Principle [1]. The other characteristic is: when we plot the set of k -dimensional points $(x_i, x_{i+1}, \dots, x_{i+k-1})$ (for all i) in the k -dimensional space, we will observe that all points fall mainly on the hyperplanes. Marsaglia [17] found that when taken in pairs, triplets, or n -tuples, the random numbers often fall on only a few planes in n -space. There is actually more than one set of parallel hyperplanes if viewing the k -dimensional space from different orientations.

2.3 Multiple Recursive Generators: MRG

As mentioned before, reliable generators with long periods are in demand for many modern applications [13]. MRGs is a popular class of such generators, which is based on the higher order recursion:

$$X_i = (\alpha_1 X_{i-1} + \dots + \alpha_k X_{i-k}) \bmod M, \quad i \geq k.$$

For a given prime modulus M , such generators can achieve the maximum period of length $M^k - 1$. And good ones have much better structural properties than the simple MLCG with the same modulus, while being almost as fast to compute and easy to implement as an LCG [12].

We say that a RNG has a “ s -distribution property”, if every s -tuple of numbers appears exactly the same number of times, with the exception of the all-zero tuple that appears one time less. In that case, LCG is 1-distributed while MRG with maximum period of $M^k - 1$ is k -distributed.

Similar to LCGs, the set of all overlapping s -tuples of values (U_i, \dots, U_{i+s-1}) , in which $U_i = X_i/M$, forms a lattice structure in $[0,1]^s$ (for details, see [2, 7, 8, 14]). MRGs of order k behave in R^{sk} like LCGs in R^k . The lattice structure may be analyzed by the spectral test. Recent computer searches for good parameters can be found in [10, 20]. Further theoretical properties of MRGs are discussed in [9].

2.4 Fast Multiple Recursive Generators: FMRG

Seeing the fact that the computing time of an MRG is about k times slower than that of an LCG, Deng and Lin [3] proposed to set as many terms of α_i in MRG to be 0 or ± 1 as possible to improve the computing efficiency. They especially proposed a random number generator called FMRG, which is a special form of the MRG with the maximum period of $M^k - 1$. The basic form of the FMRG is

$$X_i = (BX_{i-k} \pm X_{i-1}) \bmod M, \quad i \geq k.$$

It generates a new number from the last generated number and the number generated k steps earlier. The detail of searching FMRG- k is given in [3]. It is easy to see that the computing time of LCG and FMRG is almost the same since it takes the same time to add a constant increment C in LCG as to add a variable increment X_{i-1} in FMRG. Compared to a general MRG, an FMRG is fast because it requires only one multiplication and one addition/subtraction. Deng and Lin [3] also proposed to restrict the multiplier B (say, $B \leq \sqrt{M}$) for efficiency and portability.

2.5 A System of Generators by Deng & Xu: DX

A necessary condition for an MRG to have a good lattice structure is that the sum of squares of coefficients α_i is large [15]. FMRG only has two nonzero coefficients B and ± 1 . Therefore, it seems necessary to choose a large value for B or to add more nonzero terms to obtain a good lattice structure [4].

Extending the idea of FMRG, Deng and Xu [4] proposed a system of High-dimensional uniformly, Efficient, Long-cycle, and Portable uniform random number generators called DX- k - s . DX- k - s is an MRG with conditions that its corresponding primitive polynomial has s terms of common multiplier $\alpha_i = \pm B$ and the rest of $\alpha_i = 0$. They

also required the indices of nonzero terms are about $k/(s-1)$ apart. Let $[x]$ denote the integer part of a real number x . Then the nonzero coefficients can be written as

$$\alpha_1 = \alpha_{[k/(s-1)]} = \alpha_{[2k/(s-1)]} = \dots = \alpha_{[(s-2)k/(s-1)]} = \alpha_k = B$$

and the general form of DX- k - s is

$$X_i = B(X_{i-k} + X_{i-[(s-2)k/(s-1)]} + \dots + X_{i-[k/(s-1)]} + X_{i-1}) \bmod M, \quad i \geq k.$$

One big difference between FMRG and DX is that DX requires a common coefficient B for each term. The reason for such a choice is that a computer multiplication consumes much more time than an addition or subtraction. By requiring a common coefficient in DX, it only needs one multiplication so that high efficiency can be achieved.

The following are FMRG and some DX- k - s :

1. FMRG- k ($\alpha_1 = 1, \alpha_k = B$)

$$X_i = (BX_{i-k} + X_{i-1}) \bmod M, \quad i \geq k.$$

2. DX- k -2 ($\alpha_1 = \alpha_k = B$)

$$X_i = B(X_{i-k} + X_{i-1}) \bmod M, \quad i \geq k.$$

3. DX- k -3 ($\alpha_1 = \alpha_{[k/2]} = \alpha_k = B$)

$$X_i = B(X_{i-k} + X_{i-[k/2]} + X_{i-1}) \bmod M, \quad i \geq k.$$

4. DX- k -4 ($\alpha_1 = \alpha_{[k/3]} = \alpha_{[2k/3]} = \alpha_k = B$)

$$X_i = B(X_{i-k} + X_{i-[2k/3]} + X_{i-[k/3]} + X_{i-1}) \bmod M, \quad i \geq k.$$

Deng and Xu [4] conducted a complete search for DX-102- s and DX-120- s with $s=1, 2, 3, 4$ for $0 < B < \sqrt{M}$. All generators (many choices of B) of DX-102- s and DX-120- s with different s are provided on their web site [5].

A particular generator, DX-1511-4 with $M=2^{31} - 55719=2147427929$ and $B=521816$, was mentioned in [4] :

$$X_i = 521816(X_{i-1511} + X_{i-1007} + X_{i-503} + X_{i-1}) \bmod 2147427929, \quad i \geq 1511.$$

This generator has a very long period of $M^{1511} - 1 \approx 10^{14100.5}$ and equi-distributed up to 1511 dimensions.



Chapter Three

An Empirical Study

3.1 Introduction

The main objective of this study is to investigate the empirical performance of the DX- k - s generators. We would also like to know how the parameters (k , s , B) affect the empirical performance of DX- k - s generators. In this chapter, we will describe our program and the design of experiments for our empirical study.

3.2 C program

A C code for the implementation of DX-120 is provided by Deng and Xu at the website: <http://www.cs.memphis.edu/~dengl/dx-rng/>. Only slight changes on their C program are needed for implementing DX-102 and FMRG with different values of k . Appendix I lists the C code used in our study.

All tests in Diehard suite require users to provide a large binary file of 32-bit integers to test. However, the random numbers produced by the original C code of Deng and Xu are floating point numbers in the interval of $(0, 1)$. We multiply these numbers by 4294967296 ($= 2^{32}$) to produce a long sequence of 32-bit integers to create a binary file for Diehard.

3.3 Initial Seeds for 5 Replicates

In this study, we generate 5 replicates for each testing condition. Every generator needs an initial seed to start the process of producing random numbers. We use the popular LCG with $B=16807$, $C=0$, $M=2^{31}-1$ to produce 5 numbers to serve as the initial seeds for 5 replicates. We take 12345 as an initial seed for the LCG to produce the other 4 values: 770088852, 1888542194, 739539393, and 1037150863. There is no particular reason for using these 5 initial seeds, but we believe that the behavior of a good random number generator would not be affected by initial seeds.

3.4 Selection of B's

Deng and Xu [4] argued that the DX- k - s generators using large B should have a better

theoretical property and, hopefully, a better empirical performance. We choose some small and some large B 's for each DX- k - s generators in our experiments. Deng and Xu [5] have provided many values of B for each class of DX generators. For each of DX-102- s and DX-120- s , $s=1,2,3,4$, 10 different RNGs are selected, 5 RNGs with larger B and the other 5 RNGs with smaller B . For each FMRG (with $k=2, 3, 4$), we select 10 RNGs in the same manner. Then we have 110 RNGs in total. The listings of the selected B values are given in Table 1.

3.5 KS-test in Diehard Test Suite

KS-test is the Kolmogrov-Smirnov test for determining if two distributions differ significantly. The KS-test has the advantage of making no assumption about the distribution of data. It is non-parametric and, therefore, distribution free. Essentially, any test in Diehard suite tries to determine whether a set of N real numbers is randomly drawn from a uniform distribution. The KS-test in Diehard suite calculates the distance between the empirical and theoretical distribution functions and returns the probability associated with the observed value of the Anderson-Darling Statistic. It is a modification of the original KS-test and gives more weight to the tails than does the original KS-test. For most people, it is sufficient to know how evenly the generated random numbers are distributed. We denote this returned probability by KSTEST in this paper.

3.6 Tests in Diehard Suite

We run all 15 tests in Diehard suite as listed below:

1. Overlapping Sums Test (Test 1)
2. Runs Test (Tests 2a, 2b, 2c, 2d)
3. Random Spheres Test (Test 3)
4. Parking Lot Test (Test 4)
5. Birthday Spacings (Test 5)
6. Count the 1's in Specific Bytes (Test 6)
7. Ranks of 6x8 Matrices (Test 7)
8. Ranks of 31x31 and 32x32 matrices (Tests 8a, 8b)

9. Count the 1's in a Stream of Bytes (Tests 9a, 9b)
10. Monkey Tests on 20-bit Words (Test 10)
11. The Craps Test (Tests 11a, 11b)
12. Minimum Distance Test (Test 12)
13. Overlapping Permutations (Tests 13a, 13b)
14. Sparse Occupancy Tests OPSO, OQSO, DNA (Tests 14a, 14b, 14c)
15. The Squeeze Test (Test 15)
16. Summary (Summary Test)

Every test returns some p-values and at least a summary KSTEST value. Some tests (e.g., Tests 4, 6, and 14) only display several p-values but no KSTEST. So we wrote a simple program to combine the displayed p-values into a KSTEST value [see Appendix III]. This simple program is a part of the original C code in Diehard where it is used to calculate the KSTEST and returns p-values to determine whether those p-values follow the specified distribution or not. We have verified the correctness of the program by the p-values and KSTEST of several tests produced by Diehard. Some tests have more than one KSTEST value, because these tests have subtests or repeat twice. Hence, there will be in total 25 KSTEST values collected when testing a sequence of numbers.

We collect the 25 KSTEST values for each testing condition. Then we use S-plus to perform analyses of variance on these KSTEST values to study the effects of the three parameters, k , s , and B , on the performance of the RNGs.

3.7 Design of the Experiment

With the KSTEST values collected for each RNG and each testing condition (i.e., seed/ B) under study, we can perform the following analyses to study the effects of the design parameters, k , s , and B .

(1) For DX-102- s (or DX-120- s) generators, we can study the effects of s and B . The factor s has 4 levels, 1, 2, 3, 4, while B has two levels, small and large. The design is a 4×2 full factorial design. The data consist of the results of 15 tests with 5 replicates for each of the 8 ($=4 \times 2$) treatments.

(2) For FMRG- k generators, we can study the effects of the factors k and B . The design is a 3×2 full factorial design with 3 levels for factor k ($=2, 3, 4$) and two levels for factor B (small and large).

(3) Combining the data of DX-102- s and DX-120- s , we can study the effects of three factors, k , s , and B . The design is a $2 \times 4 \times 2$ full factorial design, where factor k has two levels (102 and 120), s has four levels (1, 2, 3, and 4), and B has two levels (small and large).

(4) Combining all the KSTEST data, which now consist of the results of 11 RNGs (4 DX-102- s , 4 DX-120- s , and 3 FMRG- k generators). We can study the effects of two factors, RNG and B . The design is clearly an 11×2 full factorial design.

In Chapter 4, we will report the results of our experiments and analyses.



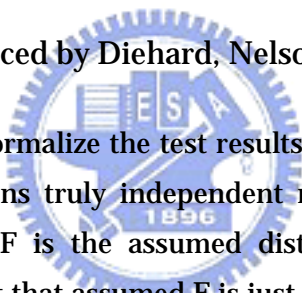
Chapter Four

Results of the Empirical Studies

A good random number generator should have some nice theoretical property and they should be verified through rigorous empirical tests. A collection of statistical tests used to evaluate random sequences was given in Diehard test suite, which was developed by George Marsaglia (see, <http://stat.fsu.edu/pub/diehard>). The original program was written in FORTRAN and it was not very portable. DiehardC 1.03, created by Scott Nelson [19], is a re-implementation of the original Diehard and is written in C. We use DiehardC 1.03 to test our random number generators and the results are reported in this chapter.

4.1 Tests of Random Numbers and Our Experiment Results

To interpret the results produced by Diehard, Nelson [19] stated that



“A p-value is an attempt to normalize the test results. A p-value should be uniform on $[0, 1)$ if the input file contains truly independent random bits. Those p-values are obtained by $p=F(X)$, where F is the assumed distribution of the sample random variable X —often normal. But that assumed F is just an asymptotic approximation, for which the fit will be worst in the tails. Thus you should not be surprised with occasional p-values near 0 or 1, such as .0012 or .9983. When a bit stream really FAILS BIG, you will get p's of 0 or 1 to six or more places. By all means, do not, as a Statistician might, think that a $p < .025$ or $p > .975$ means that the RNG has "failed the test at the .05 level". Such p's happen among the hundreds that DIEHARD produces, even with good RNG's. So keep in mind that "p happens".”

All the DX random number generators pass almost all the tests in Diehard in general. The descriptions of the 15 tests are given in Appendix III and their test results are given in Tables 3-25. Those p-values are the KSTEST values described above. B_1 to B_{10} , of each DX are the $10B$ s given in Table 1 ranked from small to large.

The KSTEST values in Tables 3-25 seem spread on the interval $(0, 1)$. Note that, we omit the table for the test of “Count the 1's in a Stream of Bytes” (i.e., Tests 9a and 9b), because all KSTEST values produced by this test are all equal to 1. By the description of

Diehard, it is usual to see few occurrences of 0 or 1 even when the sequence is random and uniformly distributed. For the whole Diehard test suite, DX can be considered as “pass” (see the results for the Summary test); but for this particular test, DX fails. This phenomenon can be explained as follows.

Notice that the “Count the 1’s in a Stream of Bytes” test is specifically designed for 32-bit generators (see Appendix III). DX random number generator with modulus $2^{31}-1$ is only capable of producing 31 random bits and the least significant bit will be always 1 when treated as a 32-bit integer. Therefore, this particular test will definitely fail the DX random number generators with modulus $2^{31}-1$. Many kinds of adjustments for these non-32-bit RNG are described by Nelson [19]. One simple remedy is to multiply the floating point by $2^{32}-1001$ (=4294966295) instead of 2^{32} (suggested by Diehard) so that DX can produce some 0 and 1 in its least significant bit. Since the test results of DX RNGs by Diehard suite are similar, we have only re-performed the DX-120- s , $s=1, 2, 3, 4$ with multiplier $2^{32}-1001$ and seed 12345 in this test. Indeed, the returned KSTEST values, provided in Table 47, are not all 1 anymore and indicate that the RNG. passes this test. In fact, the role of the multiplier is to fit the requirement of Diehard, it does not affect the essence of RNGs.

Tables 3-25 give the KSTEST values of all the tests we perform in our study. We count the frequencies of the KSTEST values less than 0.05 for each test and RNG. Figure 1 and 2 give the bar charts of these frequencies grouped by tests and RNGs, respectively. It is observed that the “Overlapping Permutations” test (Tests 13a and 13b) fails RNGs more frequently than other tests. The Overlapping Permutations test is also designed for 32-bit RNGs. This may be the reason why this test fails the DX RNGs more frequently.

Test 16 is a summary test. It collects 234 p-values produced by Diehard, and test them by a Kolmogrov-Smirnov test. Figure 1 shows that there is no of KSTEST value less than 0.05 in summary test (Test 16). In this sense, DX RNGs have passed Diehard test suite. From the results of our study, the overall performance of DX RNGs is good.

4.2 ANOVA & Discussion

To investigate if the performance of RNG may be affected by its design parameter, we collect the KSTEST values to run the four analyses of variance described in Subsection 3.7. The ANOVA reports of each analysis are provided in Tables 41-45. From these ANOVA tables, we can see that there is no apparent evidence that any factor affects the KSTEST

values with $k=102, 120$, and $s=1, 2, 3, 4$. In other words, the performance of DX random number generators in Diehard test suite is not affected by the parameters $k=102, 120$, $s=1, 2, 3, 4$, and B =small, large. The results also indicate that DX-102- s , DX120- s , and FMRG- k have similar performance when tested by Diehard suite.

4.3 A Comparison Study with LCG

To compare DX with the commonly used LCG, we also run the Diehard test suite on LCG. To generate 5 random sequences, we use the same 5 initial seeds. B s (7, 11, 14, 22, 28, 16807, 16810, 16812, 16814, 16820, 46259, 46260, 46266, 46267, 46268) are chosen and divided into three groups (small, medium, and large). The resulting KSTEST values are given in Tables 26-40. Results show that LCG performs poorly for several of Diehard tests. Based on the “pass” definition of Diehard, LCG fails. FISH LCG with $B=742938285, 950706376, 1226874159, 62089911$, and 1343714438 were recommended by Fishman and Moore [6] because of its excellent lattice structure. We also run Diehard test suite on these 5 B s with initial seed=12345 and find out that these 5 RNGs with excellent lattice structure still fail on many tests in Diehard. The results are provided in Table 48.

Table 46, the ANOVA report of LCG, it shows that B is a significant factor for the performance. The KSTEST values of Diehard test suite are apparently affected by the B values. We remarked Tests 1, 5, 8, 9, 10, 11a, summary, have the KSTEST values equal or close to 1, so that factor B shows no effects.

4.4 DX-1511-4 with $M=2^{31}-55719$ and $B=521816$

Generator DX-1511-4 with $M=2^{31}-55719$ and $B=521816$ was particularly mentioned by Deng and Xu [4]. Deng and Xu emphasized that it has a very long period and is equi-distributed up to 1511 high dimensions. We test this RNG with the Diehard test suite. With the same 5 initial seeds, the KSTEST values are given in Table 2. These KSTEST values exhibit good performance. The RNG passes all the tests. We note that DX-1511 passes the Count the 1's in a Stream of Bytes test. A reason is that it is not a generator of modulus $2^{31}-1$.

Chapter Five

Conclusion

In this information age, the computing power keeps on enhancing and the large scale simulation studies become very common. Random number generators with long period and good statistical properties become essential for many applications and simulation studies. In this thesis, we first review the DX- k - s generators proposed recently by Deng and Xu [4]. Then we produce random numbers and test them by the Diehard test suite for each RNG under study. While the empirical testing task is tedious, it is an essential step to supplement to the theoretical property and to ensure the quality of the DX generators.

In our study, DX generators with $k=102, 120$ and $s=1, 2, 3, 4$ under study perform quite well with Diehard test suite. The KSTEST values outputted by Diehard are further analyzed by ANOVA. The results indicate that the parameters, k , s , and B , do not significantly affect the performance, which means that each DX generator has similar performance with Diehard test suite. This is a good news to RNG users since there is no need to select a particular B to get a better RNG. We have particularly empirically studied the DX-1511-4 with $M=2^{31}-55719=2147427929$ and $B=521816$. This generator has a very long period of $M^{1511} \approx 10^{14100.5}$ and is equi-distributed up to 1511 dimensions. Very few RNGs pass all the tests in Diehard, and DX-1511-4 does. We have also investigated the performance of the commonly used LCG with Diehard and found that the LCGs under study perform poorly and that factor B affects the performance significantly.

While the study of random number generator is tedious and low-tech, it plays an important role in this information age. “The mechanic, who wishes to do his work well, must first sharpen his tools.” Confucius said. From the empirical testing results based on Diehard test suite, we believe that the DX generators, from $k=2$ to 1511, are good RNGs and worth a recommendation.

Reference

- [1] Bogomolny, A. Pigeonhole Principle. http://www.cut-the-knot.com/do_you_know/pigeon.shtml.
- [2] Dieter, U. 1993. Erzeugung von gleichverteilten zufallszahlen. *Jahrbuch Uberblicke Mathematik*, Vieweg, Braunschweig, 25-44.
- [3] Deng, L.Y. and Lin, D.K.J. 2000. Random number generation for the new century. *American Statistician*, 54, 145-150.
- [4] Deng, L.Y. and Xu, H.Q. 2003. A system of high-dimensional, efficient, long-cycle and portable uniform random number generators. *ACM Transactions on Mathematical Software*, April 2003, 1–11.
- [5] Deng, L.Y. and Xu, H.Q. Supplement to DX random number generators. <http://www.cs.memphis.edu/~dengl/dx-rng/>
- [6] Fishman, G.S. and Moore, L.R. 1986. An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31}-1$. *SIAM J. Sci. Comput.*, 7, 24-25.
- [7] Grothe, H. 1988. Matrixgeneratoren zur erzeugung gleichverteilter pseudozufallsvektoren. *Technische Hochschule Darmstadt*, PhD thesis.
- [8] Grube, A. 1973. Mehrfach rekursiv erzeugte zufallszahlen. *University of Karlsruhe*, PhD thesis.
- [9] Kao, C. and Tang, H.C. 1995. Symmetry property of multiplicative congruential random number generator in chi-square test. *Computer Math, Intern. J.* 55, 113-118.
- [10] Kao, C. and Wong, J.Y. 1998. Random number generators with long period and sound statistical properties. *Computers and Mathematics with Application*, 36. 3, 113-121.
- [11] Knuth, D.E. 1981. *The Art of Computer Programming: Vol 2 / Seminumerical Algorithms*. 2nd ed. Addison-Wesley.

- [12] L'Ecuyer, P. 1990. Random numbers for simulation. *Comm. ACM*, 33, 85-97.
- [13] L'Ecuyer, P., Blouin, F., and Couture, R. 1993. A search for good multiple recursive generators. *ACM Transactions on Modeling and Computer Simulation*, 3, 87-98.
- [14] L'Ecuyer, P. and Couture, R. 1997. An implementation of the lattice and spectral tests for multiple recursive linear random number generators. *INFORMS Journal on Computing*, 9, 2, 206-217.
- [15] L'Ecuyer, P. 1997. Bad lattice structures for vectors of non-successive values produced by some linear recurrences. *INFORMS Journal on Computing*, 9, 57-60.
- [16] Lehmer, D.H. 1949. Mathematical methods in large-scale computing units. *Proc. 2nd Sympos. on Large-Scale Digital Calculating Machinery*, Harvard University Press, Cambridge, MA, 141-146.
- [17] Marsaglia, G. 1968. Random numbers fall mainly in the planes. *Proceedings of National Academic of Science*, 6, 101-102.
- [18] Marsaglia, G. 1995. The Marsaglia random number CDROM including the Diehard battery of tests of randomness. <http://stat.fsu.edu/pub/diehard/>
- [19] Nelson, S. 1998. Tests for randomness. DiehardC, Version 1.03. <http://www.ciphersbyritter.com/NEWS3/RANDTEST.HTM>
- [20] Park, S.K. and Miller, K.W. 1988. Random Number Generators: Good Ones are Hard to Find. *Transactions of the ACM*, 31, 10, 1192-1201.

Appendices

Appendix I

This C program produces a binary file of random numbers used as an input to the Diehard test suite. We first generate a sequence of uniform DX random numbers using the C code provided by Deng and Xu [5], and then transform it into a 32-bit random number sequence by multiplying the numbers with a multiplier. Diehard suggested a multiplier of 2^{32} .

```
/* u_help.c : u_help generates uniform random numbers (0,1) using HELP-120-s
   For description, see Deng and Xu (2002)*/
#include <stdio.h>
#include <stdlib.h>

#define KK 120
#define PP 2147483647 /* 2^31-1 */
#define HH 1/(2.0*PP)
#define B_LCG 16807 /* for LCG */

/* _int64/quad_t:64-bit integers, which is machine dependent, see sys/types.h*/
/* use _int64/quad_t (64-bit integers) only if _int64/quad_t is defined. */
/* _int64 used in MS-C, quad_t used in GNU-C */
/* Note: operations with _int64/quad_t are faster than that with double */

#if defined(_WIN32)
typedef _int64 XXTYPE;
#define DMOD(n, p) ((n) % (p))
#elif defined(_GNUCC_)
typedef quad_t XXTYPE;
#define DMOD(n, p) ((n) % (p))
#else
#include <math.h>
typedef double XXTYPE;
#define DMOD(n, p) fmod((n), (p))
#endif

/* various B values for u_help */
static long BB1[] = {335, 1369, 1916, 2196, 2477};
static long BB2[] = {33, 283, 551, 1521, 1902};
static long BB3[] = {392, 4907, 5016, 5871, 5888};
static long BB4[] = {1441, 1651, 1906, 2543, 3075};
static int BBSIZE[4] = {5,5,5,5};
static long* pBB[4] = {BB1, BB2, BB3, BB4};

/* functions of u_help */
typedef double (*PFN_U_HELP)(void);
double u_help1(void);
double u_help2(void);
double u_help3(void);
double u_help4(void);
static PFN_U_HELP pfn_u_help[4] = {u_help1, u_help2, u_help3, u_help4};

/* internal buffer and status, initialized in su_help() */
static long BB ; /* B value */
static PFN_U_HELP pu_help; /* u_help function */
static XXTYPE XX[KK]; /* buffer */
static int II = -1; /* index */
static int IK2; /* used by u_help3 */
```



```

static int IK13, IK23;          /* used by u_help4 */

/* su_help : Initialization of help-k-s */
void su_help(unsigned int seed, int IdxS, int IdxB)
{
    int i;
    /* use LCG to initialize the sequence */
    if(seed==0) seed = 12345;      /* ensure that seed is not zero */
    XX[0] = seed;
    for(i=1; i<KK; i++) XX[i] = DMOD( B_LCG * XX[i-1], PP);

    /* initial help-k-s */
    if(IdxS <1 || IdxS > 4) IdxS = 4;
    --IdxS;      --IdxB;
    if(IdxB <0 || IdxB >= BBSIZE[IdxS]) IdxB = 0;

    BB = pBB[IdxS][IdxB];
    pu_help = pfn_u_help[IdxS];
    II = KK-1;          /* running index */
    IK2 = KK/2-1;      /* used by u_help3 */
    IK13 = KK/3-1;     IK23 = 2*KK/3-1; /* used by u_help4 */

    printf("\nseed=%d IdxS=%d, IdxB=%d, BB=%d\n", seed, IdxS+1, IdxB+1, BB);
}

double u_help1(void)
{
    int oldII = II;
    if(++II >= KK) II = 0;      /*wrap around running index */
    XX[II] = DMOD(BB * XX[II] + XX[oldII], PP);
    return ((double) XX[II] /PP) + HH;
}

double u_help2(void)
{
    int oldII = II;
    if(++II >= KK) II = 0;      /*wrap around running index */
    XX[II] = DMOD(BB * (XX[II] + XX[oldII]), PP);
    return ((double) XX[II] /PP) + HH;
}

double u_help3(void)
{
    int oldII = II;
    if(++II >= KK) II = 0;      /*wrap around running index */
    if(++IK2 >= KK) IK2 = 0; /*wrap around IK2*/
    XX[II] = DMOD(BB * (XX[II] + XX[IK2] + XX[oldII]), PP);
    return ((double) XX[II] /PP) + HH;
}

double u_help4(void)
{
    int oldII = II;
    if(++II >= KK) II = 0; /*wrap around running index */
    if(++IK13 >= KK) IK13 = 0; /*wrap around IK13*/
    if(++IK23 >= KK) IK23 = 0; /*wrap around IK23*/
    XX[II] = DMOD(BB * (XX[II] + XX[IK13] + XX[IK23] + XX[oldII]), PP);
    return ((double) XX[II] /PP) + HH;
}

double u_help(void)
{
    /* II < 0 only if su_help() is not called earlier */
    if(II < 0) su_help(0, 0, 0);
    return (pu_help());      /* call u_help */
}

```

```

}

/* sample main() function */
int main()
{
    int i, seed=1037150863, IdxB, runs=2867200, j;
    unsigned long uuu_help;
for(j=1;j<=4; j++)
{
    for(IdxB=1;IdxB<=5;IdxB++){
        static char filename[1002];
        FILE *out;

//printf("Help-120-%d, input 3 integers for: seed, runs, IdxB\n",j);
//scanf("%d%d%d", &seed, &runs, &IdxB);

su_help(seed, j, IdxB);

    printf("Input the file name :: method=%d B=%d\n",j,IdxB);
    gets(filename);

    out=fopen(filename, "wb");

for(i=0; i<runs; i++){
    uuu_help=u_help()*4294967296;
    fwrite(&uuu_help,4,1,out);
    }
}
return 0;
}

```



Appendix II

This C program is part of the source code in Diehard, and is modified to calculate the KSTEST for some tests that do not return a KSTEST value. A sample to calculate the KSTEST of DNA test (Test 14c) is produced.

```
#include <stdio.h>
#include <iostream.h>
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <malloc.h>

#define integer long
#define real double
#define ulong unsigned long
#define uint unsigned int

#define doublereal double
#undef min
#undef max
#define min(a,b) ((a) <= (b) ? (a) : (b))
#define max(a,b) ((a) >= (b) ? (a) : (b))

static FILE *output_file1, *output_file2;

#define MAX_PVALUES 240
static real pvalues[MAX_PVALUES];
static int pvalue_count = 0;

/*
   If you are using a 32 bit version of C, you don't need huge,
   and the normal malloc and free functions work just fine.
*/

#define huge

/*
   Borland C uses the non-standard farmalloc and farfree functions
*/

#ifdef __BORLANDC__
#define malloc(n) farmalloc((long)n)
#define free(n) farfree(n)
#undef huge
#endif

int acmp(const void *a, const void *b)
{
    if (*(real *)a < *(real *)b) {
        return -1;
    }
    if (*(real *)a > *(real *)b) {
        return 1;
    }
    return 0;
}
```


Appendix III

The description of the tests in Diehard suite provided in [18].

1. Overlapping Sums Test

Integers are floated to get a sequence $U(1), U(2), \dots$ of uniform $(-.5, .5)$ variables. Then overlapping sums $S(1)=U(1)+\dots+U(100)$, $S2=U(2)+\dots+U(101), \dots$ are formed. The S 's are virtually normal with a certain covariance matrix. A linear transformation of the S 's converts them to a sequence of independent standard normal, which are converted to uniform variables for a KSTEST. The p -values from ten KSTESTs are given still another KSTEST.

2. THE RUNS TEST

It counts runs up, and runs down, in a sequence of uniform $[0,1]$ variables, obtained by floating the 32-bit integers in the specified file. This example shows how runs are counted: .123, .357, .789, .425, .224, .416, .95 contains an up-run of length 3, a down-run of length 2 and an up-run of (at least) 2, depending on the next values. The covariance matrices for the runs-up and runs-down are well known, leading to chisquare tests for quadratic forms in the weak inverses of the covariance matrices. Runs are counted for sequences of length 10,000. This is done ten times. Then repeated.

3. THE 3DSPHERES TEST

Choose 4000 random points in a cube of edge 1000. At each point, center a sphere large enough to reach the next closest point. Then the volume of the smallest such sphere is (very close to) exponentially distributed with mean $120\pi/3$. Thus the radius cubed is exponential with mean 30. (The mean is obtained by extensive simulation). The 3DSPHERES test generates 4000 such spheres 20 times. Each min radius cubed leads to a uniform variable by means of $1-\exp(-r^3/30)$, then a KSTEST is done on the 20 p -values.

4. THIS IS A PARKING LOT TEST

In a square of side 100, randomly "park" a car---a circle of radius 1. Then try to park a 2nd, a 3rd, and so on, each time parking "by ear". That is, if an attempt to park a car causes a crash with one already parked, try again at a new random location. (To avoid path problems, consider parking helicopters rather than cars.) Each attempt leads to either a crash or a success, the latter followed by an increment to the list of cars already parked. If we plot n the number of attempts, versus k the number successfully parked, we get a curve that should be similar to those provided by a perfect random number generator. Theory for the behavior of such a random curve seems beyond reach, and as graphics displays are not available for this battery of tests, a simple characterization of the random experiment is used: k , the number of cars successfully parked after $n=12,000$ attempts. Simulation shows that k should average 3523 with σ 21.9 and is very close to normally distributed. Thus $(k-3523)/21.9$ should be a standard normal variable, which, converted to a uniform variable, provides input to a KSTEST based on a sample of 10.

5. THE BIRTHDAY SPACINGS TEST

Choose m birthdays in a year of n days. List the spacings between the birthdays. If j is the number of values that occur more than once in that list, then j is asymptotically Poisson distributed with mean $m^2/(4n)$. Experience shows n must be quite large, say $n \geq 2^{18}$, for comparing the results to the Poisson distribution with that mean. This test uses $n=2^{24}$ and $m=2^9$, so that the underlying distribution for j is taken to be Poisson with $\lambda=2^{27}/(2^{26})=2$. A sample of 500 j 's is taken, and a chi-square goodness of fit test provides a p value. The first test uses bits 1-24 (counting from the left) from integers in the specified file. Then the file is closed and reopened. Next, bits 2-25 are used to provide birthdays, then 3-26 and so on to bits 9-32. Each set of bits provides a p -value, and the nine p -values provide a sample for a KSTEST.

6. THE COUNT-THE-1'S TEST for specific bytes

Consider the file under test as a stream of 32-bit integers. From each integer, a specific byte is chosen, say the left-most bits 1 to 8. Each byte can contain from 0 to 8 1's, with probabilities 1,8,28,56,70,56,28,8,1 over 256. Now let the specified bytes from successive integers provide a string of (overlapping) 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's, in that byte 0,1, or 2 ---> A, 3 ---> B, 4 ---> C, 5 ---> D, and 6,7 or 8 ---> E. Thus we have a monkey at a typewriter hitting five keys with various probabilities 37,56,70,56,37 over 256. There are 5^5 possible 5-letter words, and from a string of 256,000 (overlapping) 5-letter words, counts are made on the frequencies for each word. The

quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chi-square test Q_5-Q_4 , the difference of the naive Pearson sums of $(OBS-EXP)^2/EXP$ on counts for 5- and 4-letter cell counts.

7. THE BINARY RANK TEST for 6x8 matrices

From each of six random 32-bit integers from the generator under test, a specified byte is chosen, and the resulting six bytes form a 6x8 binary matrix whose rank is determined. That rank can be from 0 to 6, but ranks 0,1,2,3 are rare; their counts are pooled with those for rank 4. Ranks are found for 100,000 random matrices, and a chi-square test is performed on counts for ranks 6,5 and ≤ 4 .

8. THE RANK TEST of 31x31 and 32x32 matrices

The leftmost 31 bits of 31 random integers from the test sequence are used to form a 31x31 binary matrix over the field $\{0,1\}$. The rank is determined. That rank can be from 0 to 31, but ranks < 28 are rare, and their counts are pooled with those for rank 28. Ranks are found for 40,000 such random matrices and a chi-square test is performed on counts for ranks 31,30,29 and ≤ 28 .

A random 32x32 binary matrix is formed, each row a 32-bit random integer. The rank is determined. That rank can be from 0 to 32, ranks less than 29 are rare, and their counts are pooled with those for rank 29. Ranks are found for 40,000 such random matrices and a chi-square test is performed on counts for ranks 32,31,30 and ≤ 29 .

9. THE COUNT-THE-1's TEST on a stream of bytes

Consider the file under test as a stream of bytes (four per 32 bit integer). Each byte can contain from 0 to 8 1's, with probabilities 1,8,28,56,70,56,28,8,1 over 256. Now let the stream of bytes provide a string of overlapping 5-letter words, each "letter" taking values A,B,C,D,E. The letters are determined by the number of 1's in a byte 0,1, or 2 yield A, 3 yields B, 4 yields C, 5 yields D and 6,7 or 8 yield E. Thus we have a monkey at a typewriter hitting five keys with various probabilities (37,56,70,56,37 over 256). There are 5^5 possible 5-letter words, and from a string of 256,000 (over-lapping) 5-letter words, counts are made on the frequencies for each word. The quadratic form in the weak inverse of the covariance matrix of the cell counts provides a chi-square test Q_5-Q_4 , the difference of the naive Pearson sums of $(OBS-EXP)^2/EXP$ on counts for 5- and 4-letter cell counts.

10. THE BITSTREAM TEST

The file under test is viewed as a stream of bits. Call them b_1, b_2, \dots . Consider an alphabet with two "letters", 0 and 1 and think of the stream of bits as a succession of 20-letter "words", overlapping. Thus the first word is $b_1 b_2 \dots b_{20}$, the second is $b_2 b_3 \dots b_{21}$, and so on. The bitstream test counts the number of missing 20-letter (20-bit) words in a string of 2^{21} overlapping 20-letter words. There are 2^{20} possible 20 letter words. For a truly random string of $2^{21}+19$ bits, the number of missing words j should be (very close to) normally distributed with mean 141,909 and sigma 428. Thus $(j-141909)/428$ should be a standard normal variate (z score) that leads to a uniform $[0,1]$ p value. The test is repeated twenty times.

11. THE CRAPS TEST.

It plays 200,000 games of craps, finds the number of wins and the number of throws necessary to end each game. The number of wins should be (very close to) a normal with mean $200000p$ and variance $200000p(1-p)$, with $p=244/495$. Throws necessary to complete the game can vary from 1 to infinity, but counts for all > 21 are lumped with 21. A chi-square test is made on the no.-of-throws cell counts. Each 32-bit integer from the test file provides the value for the throw of a die, by floating to $[0,1]$, multiplying by 6 and taking 1 plus the integer part of the result.

12. THE MINIMUM DISTANCE TEST

It does this 100 times choose $n=8000$ random points in a square of side 10000. Find d the minimum distance between the $(n^2-n)/2$ pairs of points. If the points are truly independent uniform, then d^2 , the square of the minimum distance should be (very close to) exponentially distributed with mean 995. Thus $1-\exp(-d^2/.995)$ should be uniform on $[0,1]$ and a KSTEST on the resulting 100 values serves as a test of uniformity for random points in the square. Test numbers $\equiv 0 \pmod 5$ are printed but the KSTEST is based on the full set of 100 random choices of 8000 points in the 10000×10000 square.

13. THE OVERLAPPING 5-PERMUTATION TEST

This is the OPERM5 test. It looks at a sequence of one million 32-bit random integers. Each set of five consecutive integers can be in one of 120 states, for the 5! possible orderings of five numbers. Thus the 5th, 6th, 7th,...numbers each provide a state. As many thousands of state transitions are observed, cumulative counts are made of the number of occurrences of each state. Then the quadratic form in the weak inverse of the 120x120 covariance matrix yields a test equivalent to the likelihood ratio test that the 120 cell counts came from the specified (asymptotically) normal distribution with the specified 120x120 covariance matrix (with rank 99). This version uses 1,000,000 integers, twice.

14. The tests OPSO, OQSO and DNA

OPSO means Overlapping-Pairs-Sparse-Occupancy. The OPSO test considers 2-letter words from an alphabet of 1024 letters. Each letter is determined by a specified ten bits from a 32-bit integer in the sequence to be tested. OPSO generates 2^{21} (overlapping) 2-letter words (from 2^{21+1} "keystrokes") and counts the number of missing words---that is 2-letter words which do not appear in the entire sequence. That count should be very close to normally distributed with mean 141,909, sigma 290. Thus $(\text{missingwrds}-141909)/290$ should be a standard normal variable. The OPSO test takes 32 bits at a time from the test file and uses a designated set of ten consecutive bits. It then restarts the file for the next designated 10 bits, and so on.

OQSO means Overlapping-Quadruples-Sparse-Occupancy. The test OQSO is similar, except that it considers 4-letter words from an alphabet of 32 letters, each letter determined by a designated string of 5 consecutive bits from the test file, elements of which are assumed 32-bit random integers. The mean number of missing words in a sequence of 2^{21} four-letter words, (2^{21+3} "keystrokes"), is again 141909, with sigma = 295. The mean is based on theory; sigma comes from extensive simulation.

The DNA test considers an alphabet of 4 letters:: C,G,A,T, determined by two designated bits in the sequence of random integers being tested. It considers 10-letter words, so that as in OPSO and OQSO, there are 2^{20} possible words, and the mean number of missing words from a string of 2^{21} (over-lapping) 10-letter words (2^{21+9} "keystrokes") is 141909. The standard deviation sigma=339 was determined as for OQSO by simulation. (Sigma for OPSO, 290, is the true value (to three places), not determined by simulation.

15. THE SQUEEZE TEST

Random integers are floated to get uniforms on [0,1]. Starting with $k=2^{31}=2147483647$, the test finds j, the number of iterations necessary to reduce k to 1, using the reduction $k=\text{ceiling}(k*U)$, with U provided by floating integers from the file being tested. Such j's are found 100,000 times, then counts for the number of times j was $\leq 6,7,\dots,47,\geq 48$ are used to provide a chi-square test for cell frequencies.

16. FINAL SUMMARY

Final summary test, 234 p-values collected from various tests should be another KS-test.