Pseudorandom Sequences and Random Number Generators

Elizabeth Drueke

May 9, 2016

Abstract

The generation of random numbers is vital in many computer algorithms. In this report, we look at several random number generator algorithms and determine how random we can consider them to be by performing the frequency, serial, poker, and gap tests [7].

1 Introduction

Random numbers are encountered daily in today's world. When we want to make a fair decision, we may flip a coin. When we are playing a game, we roll a die. We also daily come into contact with randomly generated numbers. Randomly generated passwords are sent to us when we sign up for yet another online account. Random numbers are used to encrypt data from our credit cards when we make a transaction. But it is important to note a distinction between these two phenomena. While a randomly generated encryption key may seem as random as the flip of a coin, or even more so, the fact is that computer-generated random numbers are only purely random if they work by measuring already naturally-occurring randomness (eg. the count of radiation in a Geiger counter from a radioactive source). Instead, the best that our technology can do today is generate pseudorandom numbers.

The need for effective random number generation is not a new one. In 1946, the first random number generator, the Middle Squares Method, was developed by John von Neumann. Motivated by his work on the hydrogen bomb, von Neumann found that he needed a computer to access a long string of randomly generated numbers, but that the computer's memory wasn't large enough to store the string internally. Thus, the Middle Squares Method was born. It is an algorithm which works as follows:

- Find a single random number (eg. a time), known as the seed, of some length n.
- Square the seed.
- Take the middle n numbers of the result as a new seed.
- Repeat as desired.

The problem here is that, of course, the sequence generated is entirely dependent on the initial seed. In particular, then, the Middle Squares Method is actually a pseudorandom number generator. This is because it is deterministic (predictable) rather than nondeterministic (uncorrelated), which a purely random sequence would be [3].

In this report, we investigate several random number generation techniques, past and present, as described in Section 2. The main goal is to determine how effective each method is at random number generation. To do this, we run these algorithms through several tests, as described in Section 3. We then discuss some applications of random number generators in Section 4.

2 Random Number Generators

As mentioned in Section 1, the only real way that a computer can generate a random sequence of numbers is by measuring an already naturally-occurring random process, something which is largely impractical (although has been tried, see [9]). The computer algorithms which have been developed to generate random numbers are therefore pseudorandom number generators. There are many fascinating properties of pseudorandom number generators, not the least of which is that, after some length of a sequence, known as the period, these sequences are expected to repeat. For example, the Middle Squares Method, discussed in Section 1, repeats once a seed is repeated. We also expect pseudorandom number generators to pass certain tests. In fact, to this end we can now introduce a precise definition of a pseudorandom number generator.

Definition 2.1: Pseudorandom Number Generator

A pseudorandom number generator is an algorithm which, starting from an initial value u_0 and a transformation D, produces a sequence $(u_i) = (D^i(u_0))$ of values in some interval. For all n, the values $(u_1, \ldots, u_n)$ reproduce the behavior of an independent and identically distributed (IID) sample $(V_1, \ldots, V_n)$ of uniform random variables when compared through a usual set of tests. [12][a]

[a] Note here that this is actually the definition of a uniform random number generator, but as we will be dealing with uniform distributions throughout this text this suffices.

As we can see, pseudorandom number generators are essentially defined by the tests used to validate their randomness. A series of such tests was developed by Kendall and Babington-Smith: the frequency test, the serial test, the poker test, and the gap test [7]. Here, we will discuss briefly what these tests are and why they are effective tests of the randomness of a sequence.

The frequency test aims to ensure that every digit occurs approximately an equal number of times in the sequence. That is, we want to ensure that the distribution is uniform. The frequency test for a sequence S over some interval $[a, b]$ is relatively straightforward: namely, ensuring that every digit occurs in the sequence approximately 10% of the time, as would be expected of a random sequence of digits. However, it is important to apply the frequency test over every segment of the sequence used to ensure that there isn't a singularly bad

section. That is, we want to split the sequence into multiple blocks and ensure that, within each block, every digit still occurs about 10% of the time. [9, 4]

The serial test is designed to check that no two digits tend to occur in a particular order. This is essentially the frequency test again, but this time ensuring all groupings of two digits occur approximately 1% of the time within the sequence. [9, 4]

In the poker test, the digits in the sequence are broken into blocks of some fixed size (eg. 5 each) and then compared with might be expected in a theoretical poker hand. In [9], the only possible hands considered were pairs, three of a kind, four of a kind, and five of a kind. We might also consider occurrences such as full houses and two pairs, as in [7]. Each of these "hands" has its own likelihood of occurring. To determine what these probabilities are, we need to determine what kind of deck we are working with. We note already that, because we can conceivably have five of a kind in a hand, it cannot be the normal 52-card deck in four suits. In fact, our deck is actually an infinite one, with no suits but 10 unique cards (one for each digit). For a hand of five cards $H = (a, b, c, d, e)$, we see that each place could be any one of the 10 cards, and so there are 50 possible hands to look at. However, to be more careful we can also look at the order in which the cards are pulled. For example, we may have a pair (a, a, b, c, d) of a pair (a, b, c, a, d), and each would count as a separate hand. So there are actually 10^5 possible hands in our game of poker.

How might we calculate the probability of one of the special kinds of hands of occurring? We will now present a couple of examples. The accepted probabilities of each of the hands is given in Table 1. The simplest example would be five of a kind, which must always take the form (a, a, a, a, a) for some $a \in \mathbb{Z}_n$[1]. As there are only 10 such values of a, there are only 10 such hands in the total of 10^5 hands, and so 5 of a kind should occur with a frequency of 0.01%.

Perhaps a more enlightening example is the case of four of a kind. Here, there are 5 possible hand combinations: (a, a, a, a, b), (a, a, a, b, a), (a, a, a, b, a), (a, a, b, a, a), (a, b, a, a, a), and (b, a, a, a, a). Now a can represent any of the 10 digits in $\mathbb{Z}_n$, and b can be any of those 10 digits except a, so there are 9 possible values of b for every a. As a result, we have a probability of four of a kind equal to

$$P(\text{4 of a kind}) = \frac{5 \times 10 \times 9}{10^5} = 0.45\%.$$

Continuing in this manner (determining the number of possible hand types which will give the desired hand and multiplying by the number of possible values for each variable), we can get the results of Table 1.

The final test suggested in [7] is the gap test, which compares the expected distance between two occurrences of the same digit in the sequence with the theoretical value. It is possible to do this for any of the 10 digits, as we would expect the result to be the same in all cases. However, for completeness, it may be better to test all 10 digits individually to ensure there are no additional biases. Again, we are faced with the question of what the theoretical value would be for frequency of a given gap size. Say we have $a \in \mathbb{Z}_n$. We would expect a to occur next in the sequence 10% of the time, as the next digit could be any of the 10 digits in $\mathbb{Z}_n$. Then 90% of the time we would have a gap of one or larger. We have a

[1]Here, $\mathbb{Z}_n = \{x \in \mathbb{Z} : 0 \leq x < n\}$.

Hand Type	Frequency (%)
5 of a Kind	0.01
4 of a Kind	0.45
3 of a Kind	7.2
2 of a Kind	50.4
Full House	0.9
Two Pair	10.8
Other	30.24

Table 1: Accepted frequencies for various poker hands [7].

gap of 1 if the sequence follows the pattern aba, which happens when the first digit is not a and the second is a. Thus, the probability of a gap of 1 is

$$P(\text{Gap Size 1}) = \frac{9}{10} \times \frac{1}{10} = \frac{9}{100} = 9\%.$$

Continuing in this manner, we see that the expected gap sizes are as found in Table 2.

Gap Size	Frequency (%)
0	10.0
1	9.00
2	8.10
3	7.29
4	6.56
5	5.90
⋮	⋮

Table 2: Accepted frequencies for various gaps between identical digits.

Throughout this discussion, we have referred to checking "how well" the sequences created by a pseudorandom number generator agrees with the accepted value. But it is important to mathematically define this notion of agreement.

2.1 The χ^2-Test

To determine how well the output of the random number generator fits the expected values of the frequency, series, poker, and gap tests, we perform a χ^2-test to determine the level of agreement. This test is designed to determine whether there is any correlation between two variables x and y.

To perform the χ^2-test, we first compute χ^2, which is given by

$$\chi^2 = \sum \frac{(v - v_e)^2}{v}, \tag{1}$$

where v is the value observed and v_e is the value expected [7].

Next, we determine the number of degrees of freedom of the problem, which is given by

$$DF = (n_x - 1) * (n_y - 1), \tag{2}$$

where n_x is the number of levels of variable x and n_y is the number of levels of variable y [1]. It is important, therefore, to note the degrees of freedom in each of the four tests used in this analysis, as shown in Table 3.

Test	Degrees of Freedom
Frequency Test	9
Serial Test	90
Poker Test	4
Gap Test	18

Table 3: The degrees of freedom of the various tests used in this analysis [7].

The final step in the χ^2-test is to consult a χ^2 table (or, in our case, an online p-value calculator [2]) to determine the p-value given the number of degrees of freedom and the computation of χ^2. This p-value gives the likelihood that differences in the observed and expected values is due to statistical fluctuations rather than a more fundamental error in the generation of the sequence. In particular, we look for p to be above a certain significance level, which we take to be 0.1 (although values of 0.01 and 0.05 are also common).

It is important to note here that in fact the χ^2-test becomes useless when the expected value of the frequency of one of the categories is less than 5. Therefore, because the highest number of degrees of freedom is 90 (for the serial test), we will restrict ourselves to looking at generated sequences of 1000 digits or more [1].

3 Testing of Random Number Generators

Often in the sciences random numbers are needed in computations (ie. in Monte Carlo simulations as discussed in Section 4). However, it is also unfortunately common that researchers do not take the time to determine how their random numbers are being generated. Such oversights can have disastrous results in the form of statistical biases that make studies unusable once discovered. This is the main motivation behind this project, in which we aim to investigate various random number generators and check their randomness using the frequency, serial, poker, and gap tests as discussed in Section 2. In addition to this, we will also look at the mean $\mu = \langle x \rangle$, ($\mu = 5$ for sequences in $\mathbb{Z}_n$) and the standard deviation σ defined by

$$\sigma = \sqrt{\langle x^2 \rangle - \mu^2}, \tag{3}$$

(given by $\sigma = 2.886$ in $\mathbb{Z}_n$) [5].

Thus, before we get into the generators used in this analysis, we must digress into a discussion of commonly used random number generators. In particular, one of the most

common types of random number generators is the linear congruential generator, defined as follows:

Definition 3.1: Linear Congruential Generator

A linear congruential generator is a pseudorandom number generator which generates a sequence of integers $(I_1, I_2, I_3, \ldots)$ in the interval $[0, m-1]$ by the recurrence relation

$$I_{j+1} = aI_j + c \mod m, \tag{4}$$

where a is what is known as the multiplier and c as the increment. [11]

Unfortunately, linear congruential generators must by definition produce sequences which eventually repeat (with a period of at most m). The idea is to optimize such a pseudorandom generator by choosing m, a, and c to make the period as long as possible. The implementation of the algorithm defined in (4) is incredibly quick, which makes linear congruential generators very widely used. [11]

We are now able to discuss the random number generators chosen for this study, many of which are based on linear congruential generators. Details of these algorithms can be found in [11]. Each algorithm is designed to output a random number $a \in [0, 1]$. We use the first digit of the decimal expansion of this number to create the string of random digits required to run the randomness checks. The exception to this discussion is the generator discussed in Section 3.5, which is the standard C++ random number generator `rand()` and `srand()`. Here, in order to get a sequnce of digits (because the number output is very large), we use modular arithmetic (mod 10).

Each of the random number generators discussed below requires a seed input (chosen to be 1000 for this study) which dictates what random number sequence will be output by the algorithm in the underlying code. The idea here is that, as mentioned before, an inherently predictable computer algorithm cannot be nondeterministic. The seed therefore helps with this by allowing a purely random number to start the sequence in some way. Within the algorithm, this seed is changed as a random number is generated, so that a new seed can be used to generate the next random number. However, whenever the same seed is used multiple times, the sequence produced will be the same. It is important that 0 is never chosen as the initial seed. When this happens, the seed never changes and the sequence generated is immediately repetitive. In particular, a non-zero seed will never, through the progression of the algorithm, yield a zero seed. [11]

3.1 Minimal Standard Generator

The first pseudorandom number generator used in this study, denoted `ran0`, is what is known as the Minimal Standard Generator, so called because it serves as a reasonable minimal standard against which to compare other random number generators. Proposed first by Lewis, Goodman, and Miller in 1969 [11], the Minimal Standard Generator is a multiplicative congruential algorithm. Simpler still than the linear congruential generator, this algorithm

relies on the recurrence relation

$$I_{j+1} = aI_j \mod m. \tag{5}$$

The algorithm used in this analysis specifically was proposed by Park and Miller with the settings $a = 7^5 = 16807$ and $m = 2^{31} - 1 = 2147483647$. The period of `ran0` is expected to be $2^{31} - 1 = 2.1 \times 10^9$. [11]

We found that this algorithm behaves fairly well. It passes the frequency and serial tests without much of a problem. The only test which is does not appear to pass is the poker test. Here, the p-value is ~ 0.04, which is far below our significance threshold.

Statistic	Value	Significance
μ	4.50159	0.04%
σ	2.87309	0.45%
χ^2 (Frequency Test)	5.68698	0.7708
χ^2 (Serial Test)	86.2657	0.5919
χ^2 (Poker Test)	10.0296	0.0399
χ^2 (Gap Test)	145.295	< 0.0001

Table 4: Results of the frequency, serial, poker, and gap tests performed on `ran0`, and the significance of those results (the percent difference from the expected value in the case of the mean μ and the standard deviation σ, and the p-value for the χ^2 tests.

3.2 Shuffling the Minimal Standard Generator

Our next pseudorandom number generator, `ran1`, uses `ran0` for its random variable but also uses a shuffling algorithm developed by Bays and Durham to remove lower-order correlations. In this shuffling, a randomly generated element of the sequence, I_j, is chosen to be output as some other element of the sequence (typically I_{j+32}). According to [11], `ran1` passes all known statistical tests excepting when it is called somewhere on the order of its period ($\sim 10^8$). [11]

We found that this algorithm passes the frequency and poker tests, but not the serial test. This is an interesting result as `ran1` is designed to help rid `ran0` of correlations, and `ran0` passed the serial test without a problem.

3.3 Double Shuffling and Recombinations

Claimed to be a "perfect" random number generator (up to floating point errors) by [11], `ran2` adds another level of shuffling to further remove the correlations existing in `ran1`, as well as other randomness checks originally proposed by L'Ecuyer. One such check is combining `ran0` and `ran1` such that the period of the sequence generated is the least common multiple of the periods of the first two algorithms. This is done by adding the two sequences produced by these algorithms modulo the modulus m of either of them. In this way, `ran2` is designed to have a very long period ($> 10^8$). [11]

Statsitsic	Value	Significance
μ	4.50043	0.01%
σ	2.87041	0.54%
χ^2 (Frequency Test)	7.19849	0.6165
χ^2 (Serial Test)	112.818	0.0522
χ^2 (Poker Test)	5.22794	0.2647
χ^2 (Gap Test)	116.184	< 0.0001

Table 5: Results of the frequency, serial, poker, and gap tests performed on `ran1`, and the significance of those results (the percent difference from the expected value in the case of the mean μ and the standard deviation σ, and the p-value for the χ^2 tests.

We found that this algorithm readily passed all of the frequency, serial, and poker tests. In particular, it also has the lowest percent difference between the observed and expected average and standard deviation of any of the three algorithms discussed up until this point.

Statsitsic	Value	Significance
μ	4.4999	$< 0.01\%$
σ	2.87382	0.42%
χ^2 (Frequency Test)	8.62035	0.4730
χ^2 (Serial Test)	93.7899	0.3714
χ^2 (Poker Test)	5.28678	0.2591
χ^2 (Gap Test)	148.062	< 0.0001

Table 6: Results of the frequency, serial, poker, and gap tests performed on `ran2`, and the significance of those results (the percent difference from the expected value in the case of the mean μ and the standard deviation σ, and the p-value for the χ^2 tests.

3.4 Fibonacci Generator

`ran3` was originally presented by Knuth and is in fact based on subtractive methods rather than acting as a linear congruential method. These subtractive methods make `ran3` a Fibonacci generator. The idea here is that the next term in the sequence comes from the subtraction of two previous terms. In particular,

$$x_k = x_{k-a} - x_{k-b} \tag{6}$$

for some fixed $a, b \in \mathbb{Z}$ known as lags. Fibonacci generators are typically very efficient and pass most randomness tests thrown at them, although they do require more storage than linear congruential algorithms. [6]

We found that, indeed, `ran3` passed all three of the frequency, serial, and poker tests, just as did `ran2`. However, this algorithm had much lower p-values for the serial and poker tests than did `ran2`, indicating that perhaps this is not the stronger of the two.

Statsitsic	Value	Significance
μ	4.50161	0.04%
σ	2.87307	0.45%
χ^2 (Frequency Test)	5.92085	0.7478
χ^2 (Serial Test)	106.29	0.1157
χ^2 (Poker Test)	7.28835	0.1214
χ^2 (Gap Test)	128.797	< 0.0001

Table 7: Results of the frequency, serial, poker, and gap tests performed on `ran3`, and the significance of those results (the percent difference from the expected value in the case of the mean μ and the standard deviation σ, and the p-value for the χ^2 tests.

3.5 C++ Random Number Generator

The final random number generator tested in this analysis is the standard `rand()` function in the standard C++ library. According to the C++ documentation, the default underlying algorithm here is another linear congruential algorithm. We chose to test this algorithm primarily as a check that the code created to run the four randomness tests was working properly, working under the assumption that, as it comes standard, it has likely been well-tested at this point.

We found that, as expected, `ran4` passed the frequency, serial, and poker tests with flying colors. However, it failed the gap test (as did the other four random number generators tested). This result leads the author to believe that the code used in the gap test is flawed somehow, and that further study is needed before any claims can be made.

Statistic	Value	Significance
μ	4.50236	0.05%
σ	2.87138	0.51%
χ^2 (Frequency Test)	7.25112	0.6110
χ^2 (Serial Test)	79.4787	0.7784
χ^2 (Poker Test)	3.55947	0.4689
χ^2 (Gap Test)	136.401	< 0.0001

Table 8: Results of the frequency, serial, poker, and gap tests performed on `ran4`, and the significance of those results (the percent difference from the expected value in the case of the mean μ and the standard deviation σ, and the p-value for the χ^2 tests.

4 Applications

Random number generators are, as mentioned in Section 1, very important in today's high-tech society. One important application which was mentioned is the application to encryption. Encryption is used daily by the average consumer; every time someone makes a

purchase with a credit card or by some other electronic means, they become vulnerable to hackers and identity theives and must rely on the encryption of their data for the safety of their information. This means that numbers must be generated in such a way that a crook would be unable to replicate them (ideally), or at least so that the crook will be unable to replicate them without a large amount of work. This means that if the period of the random number generator used in the encryption is small, the data will not be well secured. Similarly, if the seed is known, the pattern of random numbers would be able to be replicated identically. Similar situations arise in the keeping of any information, such as employee information stored by employers and security information being stored or investigated by organizations such as the NSA.

Large numbers of applications arise in the field of physics as well. For example, the Ising model in statistical physics is a simple way to model the behavior of spins in a ferromagnetic material. Ferromagnetic material is a material which can retain its magnetization even outside of the presence of an external magnetic field. The Ising model assumes that a ferromagnetic material has a lattice structure, and then defines its energy as

$$E = -J \sum_{\langle kl \rangle}^{N} s_k s_l - B \sum_{k}^{N} s_k, \tag{7}$$

where $s_k = \pm 1$ is the spin of the k^{th} lattice point, J is a coupling constant expressing the strength of the interaction between the neighboring spins, B is the externam magnetic field the material is in, and the first sum is taken over nearest neighbors (ie. lattice points directly above, below, to the left, or to the right of each other, assuming periodic boundary conditions). Because we are working with ferromagnetic materials, which can retain magnetization in the absence of magnetic fields, we can take $B = 0$, and so (7) becomes

$$E = -J \sum_{\langle kl \rangle}^{N} s_k s_l. \tag{8}$$

The Ising model is interesting to this discussion because its computer simulation involves the use of Monte Carlo, which essentially uses random variables to guess and check the actual solution to a problem. Monte Carlo is used in all areas of physics, particularly in areas where multi-dimensional integration is necessary. It is useful because the error in a Monte Carlo simulation goes as the inverse of the square root of the number of samples used in the calculation. [5]

The applications of Monte Carlo simulation are nearly endless. In a recent paper [8], Monte Carlo was used to test diffusion in cells – particularly pourous cells and brain cells, for which it is very difficult to obtain good data. In another paper [10], Monte Carlo was used to model a simple closed economy. In all cases where Monte Carlo is useful, pseudorandom number generators are vital.

5 Conclusion

Overall, we have presented here a fairly inclusive study of the properties and usefulness of various random number generators. In particular, we looked at four random number

generators defined in [11], as well as the standard C++ `rand()` function, and determined their validity as random number generators based on the four tests suggested by Kendall and Babington Smith [7], the frequency, serial, poker, and gap tests. Our results seemed reasonable, excepting the results from the gap test which were deemed inconclusive pending further investigation under the suspicion of a bug in the code used to run the test. Another very commonly used test is the calculation of the autocorrelation function R of the sequence, given by

$$R(k) = \sum_{n=1}^{\infty} (x_n - 5)\,(x_{n+k} - 5)\,, \tag{9}$$

and expected to be sequence of truly uncorrelated numbers [9]. It might be nice to incorporate this into future studies.

Pseudorandom number generators are such a vital part of our day to day experience that it is vital to understand how they work both for safety purposes but also for the purposes of furthering science and conducting research. Hopefully, this paper will motivate other scientists to look into the black box pre-written random number generators frequently become in order to better understand the biases that can arise in their use.

References

[1] Chi-square test for independence. web, 2016.

[2] P value calculator. web, 2016.

[3] B. Cruise. Pseudorandom number generators. Web. Video.

[4] F. Gruenberger. Notes. *Mathematics of Computation*, 4:244–245, 1950.

[5] M. Hjorth-Jensen. Computational physics lecture notes fall 2015, August 2015.

[6] D. Kahaner, C. Moler, and S. Nash. *Numerical Methods and Software*. Prentice Hall, 1989.

[7] M. G. Kendall and B. B. Smith. Randomness and random sampling numbers. *Journal of the Royal Statistical Society*, 101(1):147–166, 1938.

[8] H.-G. Lipinski. Monte carlo simulation of extracellular diffusion in brain tissues. *Physics in Medicine and Biology*, 35:441, 1990.

[9] T. G. Newman and P. L. Odell. *The Generation of Random Variates*, volume 29 of *Griffin's Statistical Monographs and Courses*. Hafner Publishing Company, 42 Drury Lane, London, WC2B 5RX, 1971.

[10] M. Patriarca, A. Chakraborti, and K. Kaski. Gibbs versus non-gibbs distributions in money dynamics. *Physica A: Statistical Mechanics and its Applications*, 340(1–3):334 – 339, 2004. News and Expectations in Thermostatistics.

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3 edition, September 2007.

[12] C. P. Robert and G. Casella. *Monte Carlo Statistical Methods*. Springer, 1999.