Michael Newman

Algorithms

Statistical Analysis of Pseudo-Random Number Generators

# Intro

Random number generators (RNG) have innumerable applications in computer science and occupy a great deal of academic study within the field. They can be found in lotteries, video games, political polls, and in applications requiring random sampling. These applications rely on the underlying generator to be random. Imagine if the Mega-Millions lottery never picked the number 54 as a winning number or always picked 22 as one. The lottery wouldn’t be fair and the results would be far from random. Clearly, the randomness of the underlying RNG directly impacts the validity of the application.

An algorithm is a *precise* set of steps; it cannot be made *truly* random. If you know the steps and you know any required parameters, you can recreate the exact sequence of numbers. Thus, RNGs are impossible to design without relying on a truly random input stream. However, they do exist, as we can find “true” randomness in nature.

This paper focuses on Pseudo-Random Number Generators (PRNG), close approximations of RNGs. These work by performing a series of mathematical calculations to produce a “random” sequence. The Java Standard Library (JSL) generator defined in java.util.Random is compared and contrasted with a custom PRNG designed specifically for this paper. The custom generator is developed and first, using a visual test to generate an approximation of the randomness observed. While the JSL generator is deemed to be random, the custom generator’s randomness is rejected.

# Design

Each generator works by filling an array of size 1,000,000 with random bytes. These bytes are then fed to the analysis code. In the Mono-bit Frequency and Runs tests, these bits are converted to bit strings, so the bits can be analyzed instead of the bytes. In the other tests, the bytes are turned into integers, although this process differs in each case (see below).

Reports were generated, when practical, directly from the testing code. The Coordinate Pair tests used R to generate plots, with all settings set to their defaults with the exception of the axis ranges, which were both set to (-127, 127). The Frequency distribution bar graphs were generated in Excel and then standardized to display the same range. These two tests were developed by the author; however, they represent classical statistical methods for analyzing sample spaces.

The Mono-bit Frequency and Runs tests were taken from “A Statistical Test Suite for

Random and Pseudorandom Number Generators for Cryptographic Applications,” a test suite designed specifically to analyze RNG and PRNG by the National Institute for Standards and Technology (NIST).

Links to the test code, raw results, and references can all be found in Appendix A.

# Statistical Tests

### Coordinate Pair Test

In this test, Cartesian coordinates were generated and plotted to visually represent the randomness of a sequence of integers. First, the bytes in the sequence are converted to signed integers. Then, pairs are formed by combining sequential values until all values are exhausted. For example, the sequence [1, 62, 7, -16] would generate the pairs (1, 62) and (7, -16). These points were exported to a CSV file by the test code and then plotted in R to visually examine the randomness of the sequence. A random sequence is defined as one where no visual pattern appears and the points span the entire range in both the x and y direction. When this process is performed in R, a truly random sequence presents as a square box. This is intentional, due to the size of the point being plotted and the size of the plot itself. Thus, the test for randomness is the proportion of white space. Note that this test is visual and is not designed to be autorotative by itself. Other testing methods employed in this paper are designed to calculate a mathematical determination of the randomness of the sequence.

### Frequency Histogram Test

Integers were generated spanning (0, 255) by converting each byte into an unsigned integer. The distribution of the integers was measured in two ways. First, the standard deviation was calculated to evaluate the variation of the sequence. The lower the standard deviation, the greater the randomness. However, a standard deviation of 0 is not expected, as uniform distributions are rarely ever *exactly* uniform. Second, the interval was split up into 8 evenly sized ranges, each spanning 32 consecutive integers. The observations were totaled over this range and then plotted as a frequency histogram to visually estimate the randomness of the sequence.

A truly random number sequence would appear almost perfectly distributed across the range of possible values. However, the probability of it being *exactly evenly* distributed across the range is incredibly low. Thus, approximate equality is the measure of randomness employed in this test.

### Mono-bit Frequency Test

The simplest test used within this paper examines the frequency of a PRNG by investigating the ratio of 1’s and 0’s. While NIST lays out a novel approach to implement this strategy, java.util.BitSet offers a cardinality method which returns the amount of set bits in a sequence. Thus, the strategy defined by NIST on page reference 2-2 was modified to use this metric. As NIST explains, “[t]he purpose of this test is to determine whether the number of ones and zeros in a sequence are approximately the same as would be expected for a truly random sequence.” A “p-value” is generated which measures the randomness of the sample. Any p-value bellow 0.01 signifies that a sample is non-random.

### Mono-Bit Runs Test

In this test, randomness was examined through analyzing how the sequence oscillates between 1’s and 0’s. Per NIST, “[t]he focus of this test is the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits” (2-5). Implementation details can be found on page reference 2-6 in “A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications.” This test also generates a “p-value” with the same cut off as the Mono-bit Frequency Test.

# Custom Generator

The custom random number generator was developed based on code from Chris Smith, a software developer from google. The generation sequence begins with a seed value, a pre-determined, non-random value set by the user of the program. This value is multiplied by a *multiplier*, chosen to be a large prime number still within the integer space. This value becomes the new seed value, and the calculation continues. The new seed is then divided by a *mod­\_value*, chosen to be a prime number significantly larger than the required interval (-127, 127). The remainder of this calculation becomes the *int\_ran* value. This value is then normalized to the range (0, 254) by extracting the remainder of the previous result divided by 254. To generate a value within the specified interval, 127 is subtracted from the previous result. This *normalized\_ran* value is then converted into a byte. This process is summarized in the pseudocode provided bellow.

**Pseudocode**

ran(seed, multiplier, mod\_value):

seed = multiplier \* seed % mod\_value

int\_ran = seed % 254

normalized\_ran = int\_ran – 127

return (byte) normalized\_ran

## Method Development

Various values for the multiplier, mod\_value, and initial seed were examined to determine their impact on the randomness of the generated sequences by the custom generator. In this stage, the Coordinate Point Test was employed to understand the effect of different values on this sequence.

**Test 1**

|  |  |
| --- | --- |
| **Initial Seed** | 1 |
| **Mod\_Value** | 101 |
| **Multiplier** | 7 |

A screenshot of a cell phone

Description automatically generated

The first set of numbers were suggested by Smith in his explanation of random numbers. However, in his explanation he states that the *mod\_value* chosen should be much larger than the actual range of the size. Furthermore, the multiplier chosen doesn’t generate numbers large enough to actually span the range. The effects of this are clear in the plot. Since the number never gets large enough, when the normalization step is performed it instead generates numbers exclusively in the 3rd quadrant.

**Test 2**

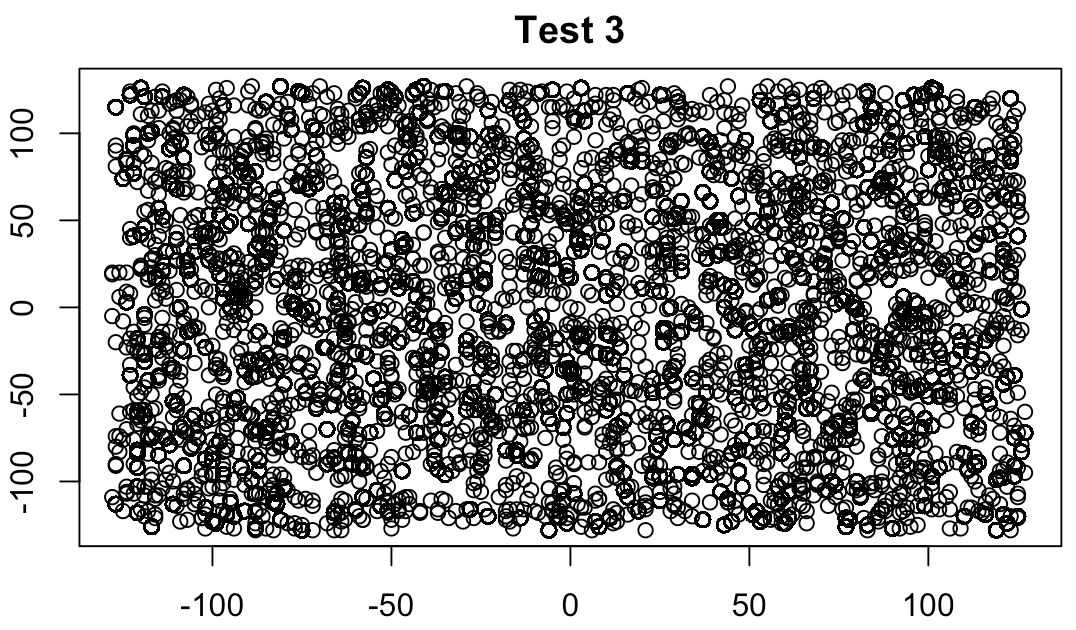
|  |  |
| --- | --- |
| **Initial Seed** | 1 |
| **Mod\_Value** | 2,475,091 |
| **Multiplier** | 134,520,263 |

**A screenshot of a cell phone

Description automatically generated**  For the second test, two large prime numbers were chosen for the multiplier and the mod\_value. As demonstrated by the plot, this increased the randomness of the sequence. However, it remains clear that this sequence is not random. While the numbers are more evenly distributed across the interval, all numbers within this interval were not present.

**Test 3**

|  |  |
| --- | --- |
| **Initial Seed** | 137,363 |
| **Mod\_Value** | 2,475,091 |
| **Multiplier** | 134,520,263 |

****

Due to the success of the previous values, the mod\_value and multiplier were kept the same. Instead, the effects of changing the initial seed value were examined. As demonstrated visually above, using a larger seed increased the entropy of the coordinate points. The points appear evenly distributed in the x and y planes and show no clear, discernable pattern. Further, the amount of white space has substantially decreased from the first trial. Thus, this set of parameters were chosen to undergo further study and comparison against the JSL PRNG.

### Possible Improvements

While these parameters are considered sufficient for this analysis, further studies could absolutely improve the quality of this PRNG. This paper examines 3 combinations of parameters and chooses, explicitly, to stop there. Further studies should examine other possible combinations of prime numbers which produce greater and lesser amounts of entropy. This algorithm also relies on a fixed initial seed value, which means that the sequence produced is the same every single time this code is ran. The effects of other seed-generation methods are not examined by this paper. This was a specific experiment design choice, as the ability to replicate the precise sequence of numbers multiple times enables any reader to recreate the exact sequence used for examination. However, the above sequence of trials suggests that there is a strong correlation between the randomness of the sequence and the value of the initial seed. Thus, further study is needed to examine seed generation methods and their impact on the entropy of the sequence.

# Results

The results of the test suite are examined bellow. For raw results see Appendix A.

### Coordinate Plane Test

A screenshot of a computer

Description automatically generated A screenshot of a cell phone

Description automatically generated

For this test, the results of the coordinate plane test are compared across the JSL generator and the custom generator. The JSL results on the left match the specification for random generation *exactly*: there is no observable white space at all. Further, this is not a funciton of the size of the graph on display here. At all image sizes studied, there was no observable white space.

When the custom generator’s plot was examined earlier in this paper, it was considered to be random *enough* for the statistical tests performed in this paper. However, when compared directly to the JSL plot, the custom generator plot’s randomness deterioates. Clearly, the custom generator is not as random as the JSL generator. Thus, this test accepts the randomness of the JSL generator and rejects the randomness of the custom generator.

### Frequency Distribution Test

In the second visual test, the frequencies of ranges of values were compared. In this test we get our clearest visualization of the non-randomness of the custom generator. The expected percentile of each range is 12.5%, however the range (128, 159) accounted for only 11.84% of the observed values. While a 0.66% difference is not large, it represents a significant deviation from randomness.

|  |  |  |
| --- | --- | --- |
|  | **JSL** | **Custom** |
| Standard Dev | 62.68 | 705.77 |

**Comparison of Standard Deviation by Value**

This deviation becomes even more apparent when examining the standard deviation of the observed value count. Note, this calculation is based on the count of observed values for each value, not over any range. The custom generator suffered a 1,0025.99 % decrease in randomness, measured by the standard deviation. Thus, this test accepts the randomness of the JSL generator and rejects the non-randomness of the custom generator.

### Mono-Bit Frequency Test

|  |  |  |
| --- | --- | --- |
|  | **JSL** | **Custom** |
| **Proportion of 1s** | 4,000,934 (50.0 %) | 4,001,983 (50.0 %) |
| **P value** | 0.5087 | 0.1605 |
| **Is Random?** | Random | Random |

Recall that the cut-off for a sequence to be determined to be non-random is p value bellow 0.01. By this metric, the custom generator is less random than the JSL generator as its’ value is smaller and closer to the cut-off value. Based on the examination of the Frequency Distribution Test and the Coordinate Plane Test, this is expected. The custom generator appears random, however not to the degree the JSL appears. In this test, both sequences were determined to be random according to the standards set by NIST.

### Mono-bit Runs Test

|  |  |  |
| --- | --- | --- |
|  | **JSL** | **Custom** |
| **P-Value** | 0.6255 | 9.4508 x 10-29 |
| **Is Random?** | Random | Non-Random |

In our final statistical test, we find our most conclusive evidence of the non-randomness of the custom generator. In fact, the custom generator is more than 28 \* 10-29 away from being considered random by the standards set by NIST. Recall, this test examines the oscillation between 0’s and 1’s by examining the amount of sequential, identical bits. According to NIST, such a low p value suggests that the oscillation is happening too fast for the sequence to be random. Thus, this test accepts the JSL generator and rejects the custom generator.

# Conclusion

The true randomness of a PRNG should always be considered when choosing an implementation when developing software. There are some applications, such as random sampling or the lottery, which require incredibly high levels of confidence in the randomness of a generator. On the other hand, some applications, such as backtracking algorithms, do not require as much certainty. This paper does not seek to label one generator or the other as a “winner.” Instead it only seeks to examine the randomness of the algorithms. If true randomness is a chief design concern, the JSL generator is highly recommended. However, if an approximation of randomness suits the design needs of an algorithm, then the custom generator developed here would suffice.

Throughout this paper, the randomness of the JSL random generator and a custom random generator have been examined. The first test, the Coordinate Plane visual test was used to approximate the randomness of the generators. While the custom generator appeared to be random when examined in isolation, comparison with the JSL generator demonstrated its’ lack of randomness. Thus, the first test rejected the randomness of the custom generator and accepted the randomness of the JSL generator. Next, frequency histograms of the two generators were examined. Yet again, the non-randomness of the custom generator was exposed while the JSL generator was deemed to be random. Third, the frequency of 1’s and 0’s was examined by the Mono-bit Frequency Test. In this test, both generators were considered to be random. However, the statistics used to generate that evaluation demonstrated that the custom generator was still not as random as the JSL generator. Finally, the amount of bit runs throughout the sequence was studied. In this test, the JSL generator was again determined to be random while the custom generator was determined to be non-random. Thus, this paper concludes that the JSL generator is random while the custom generator is not.

Appendix A

The results generated by the test code for the JSL generator can be found at

<https://github.com/Newms226/PRNG-Analysis/blob/master/logs/jsl_results.txt>

Results for the custom generator can be found at

<https://github.com/Newms226/PRNG-Analysis/blob/master/logs/custom_results.txt>

The source code for this project can be found at

<https://github.com/Newms226/PRNG-Analysis>

The R notebook used to generate all plots can be found at

<https://github.com/Newms226/PRNG-Analysis/blob/master/r_file.Rmd>

The Excel document used to create the frequency distributions can be found at

<https://github.com/Newms226/PRNG-Analysis/blob/master/frequency.xlsx>

Appendix B: Works Cited

Rukhin, Andrew, et al. “A Statistical Test Suite for Random and Pseudorandom Number

Generators for Cryptographic Applications.” 2000, NIST, doi:10.6028/nist.sp.800-22.

Smith, Chris. “Build Your Own Simple Random Numbers.” Sententia Cdsmithus, 10 Oct. 2011, cdsmith.wordpress.com/2011/10/10/build-your-own-simple-random-numbers/.