Comparison of different Pseudo Number generation methods.

Thomas Kyte

University of South Carolina Upstate

Computer Science Senior Seminar

January 30th, 2020

[tkyte@email.uscupstate.edu](mailto:tkyte@email.uscupstate.edu)

ABSTRACT

In this paper, I am comparing different implementations of pseudo-random number generation. Pseudo number generation is used in a variety of applications when the output needs to appear unpredictable. Computers themselves are not good at giving random output, so algorithms are used to provide the appearance of randomness. Implementation through algorithms has the benefit of being deterministic, which makes testing more reliable; however, if the Algorithm is too simple, it can be reverse-engineered. As such, PRNG is a potential venerability in security applications. A good generator is sufficiently complex to give inherently random output while maintaining speed and being deterministic, allowing developers to get consistent output during testing. This paper aims to compare several implementations of pseudo number generation and compare them based on speed and randomness. These metrics will also be compared when generating arrays of different sizes.

Keywords

Deterministic, Pseudo-random number generation, PRNG, Seed, TRNG, chi-squared, p-value

Copyright info goes here

1. INTRODUCTION

Pseudo-random generation is used in a wide variety of applications, from online gaming to cryptography. However, it can commonly be mislabeled as random number generation (RNG). RNG is a general term used for true random number generation (TRNG) or pseudo-random number generation. It is crucial to define PRNG and its use cases to understand and compare Pseudo-random number generation or PRNG fully. PRNG is any formula implemented to produce a sequence of seemingly random numbers. These so-called random numbers result from taking a seed number and plugging it into the PRNG algorithm. A seed is the starting number used when starting number generation, and the seed ends when the Algorithm loops back to the starting number or consistently gives the same output. Therefore, using set seed numbers when comparing PRNG implementations is essential as smaller seeds result in reduced number variation. In order to compare different PRNG implementations fairly, testing must occur at multiple output sizes and seed sizes to reflect the seed's impact on the resulting output.

This paper will analyze three different methods of PRNG, comparing the results based upon the time taken to run, how random the results appear, and the dataset coverage. The first method is using a Linear Congruential Generator. The second method is a Lagged Fibonacci generator. The third method is the Middle Square method.

The variation in numbers and the speed at which these algorithms run is vital for determining when to use different methods and if the technique is random enough to use in a real-world application. The methods chosen represent two popular methods and one legacy method used to compare randomness and see if algorithms like the middle square method still have a place in modern programming.

1. LITERATURE REVIEW

Recent research on random number generation emphasizes TRNG and how computers can generate more random output. The main difference in these generation methods is that while PRGN uses a seed as the source, TRNG uses a physical component like noise from a component as its source1. One such version of TRNG uses a specialized ring oscillator (RO) set into a metastable mode allowing for TRNG with only digital components and no specialized design1. This method has passed Federal information processing standards showing it is sufficiently random. Another branch of RNG research is TRNG based on RO but relies on post-processing to meet security standards. The researchers referred to this method as chaos-based post-processing and used a logistic map to remove stream correlation2. This method resulted in increased security at the cost of increased energy usage. While research into TRNG benefits security and gambling applications, it is disadvantageous when developing simulators and other programs where data needs to be random yet consistent over many trials.

Researchers performed a similar analysis to this paper in 2020 focused on the reliability of PRNG based on visual testing3. However, this research did not compare these PRNG methods and instead concentrated on using PRNG in stochastic processes concluding that computing stochastic processes with perfect confidence is unlikely to exist using PRNG. In 2015 a member of Syiah Kula University showed how to improve the design of linear congruential generators4. In 2017 a Cornell student published a paper on combining a Weyl sequence with the middle square method to overcome the issue of repeating 0's in the series after the first occurrence5. This research highlights a lack of research comparing PRNG as most papers focus on improving one method or on TRNG. This paper aims to fill the gap and provide information comparing different PRNG implementations for non-security purposes.

1. Methodology

When comparing PRNG the seed and variables chosen have a massive impact on the number of unique integers before a cycle is formed. This variability leads to the challenge of avoiding short cycles without cherry-picking ideal results. This issue was mitigated by having all algorithms receive the same seed of 39991234 where none of the methods suffered from entropy. To further compare methods three more seeds were chosen at random to compare general performance of each Algorithm. It is important to note here that reusing the same seed can bias effects in simulations8. This paper only tests with four seeds for time scale so there can be some bias in time output and many more runs and seeds are needed if 100% certainty is needed on time comparison. A limited number of seeds were used due to time constraints of this research.

The first PRNG is the Linear Congruential generator. The Algorithm for this is: rn+1=a\*rn + c (mod m)

In this Algorithm, A represents a constant multiplier, Rn is the seed, and c is a constant value added. Modular division is performed on the result to bring it into the range of random numbers we want to generate. This Algorithm has many variables making it more susceptible to looping, but the length of unique numbers generated is increased by choosing a constant c which has no factors in common by the modulo m. This approach works, but testing different A and C values are essential for developing an effective generator using the Linear Congruential method.

The next PRNG is the Lagged Fibonacci generator. This algorithm is represented by: Sn = Sn-j ★ Sn-k (mod m), 0<j<k. This Algorithm works by turning the seed into an array then taking the numbers at j and k and performing an operation on them. The Star is used in the formula to show that any operation can be used there. For the purposes of this paper, the star represents addition. The main challenge to this generator is choosing the operation that will provide the most unique numbers from the given seed.

The final PRNG being tested is the middle square method. In this Algorithm, the seed is squared, and then the middle digits of the resulting number become the new seed. This Algorithm is especially susceptible to repeating as once the middle digits are zeros every number after that will be zero as well is it is important to choose a seed where it loops without ever having a zero or having a significant length where the inviable collapse of the Algorithm isn’t an issue.

1. Implementation

This experiment is fully done in Python3 using a main method with six helper functions.

The main method consists of a series of function calls for each PRNG to get initial data sets. Then a function is called to generate runtimes with a follow-up to calculate the average time of 50 runs. The main method concludes with calls to calculate the chi-squared values, percent coverage, and output graphs showing each Algorithm's output.

The first function call used is linear\_congruential, which is just the formula stated in the methodology and returns the numbers generated as an array. For this test the constant A is 1526 and the constant C is 57.

The next helper function is lag\_fib which is a lagged Fibonacci generator using addition. The J and K values are 3 and 7 respectively. This function also returns all numbers generated as an array.

The final helper function is mdl\_sqr which preforms the middle square method. The minimum seed length is 12 and 0 is appended to the start and end of the seed to pad it until it reaches the required length. To give a longer run of end numbers the middle for this middle square method is considered to be the digits three through nine in the seed. Like the other two methods, the generated numbers are returned as an array.

The comparison of how random the output of each method is done using SciPi which is an opensource tool for scientific computing. The system used for these tests is an Inspiron 16 plus with 11th Gen Intel i7-11800H and 16GB of RAM.

1. dataset and experiment setup

The data for this comparison is split in three parts: time, randomness, and percent coverage. The time metrics are based of the average time taken after 50 generations of each Algorithm.

Randomness is determined by the chi-squared method and measured at each different length. The chi-squared method compares expected vs actual results. For example, if we generate one thousand numbers between zero and one thousand, each number would be expected to appear roughly once. Since PRNG loops, the actual output will vary from the expected. Chi-squared compares the actual output with expected output and then gives the likelihood for this data to be random at different confidence levels.

Originally, the data set went to 100,000,000 but that was reduced to 10,000,000 to be more manageable array size, allowing for more efficient testing.

The final data component is percent coverage which is simply the count of unique numbers generated divided by the range of possible output. The output from each Algorithm and size is also output as a graph to quickly identify output bias or how random each result is compared to each other.

1. Result analysis (chi-squared)

Using the chi-squared method, actual results can be compared with expected results. If the results were truly random, then each number should occur roughly the same number of times for the chi-squared method; this assumes exactly the same number of times for each possible output. The P-value result from the chi-squared method shows how statistically likely that output is a result of one means that the actual and expected outputs are exactly the same. The smaller the P-value is the less likely that result is. If the P-value is less than .05, the result is significantly different at this value, there is only a five percent chance that the conclusion is incorrect about the results being random.

* 1. Linear congruential chi-squared results:

Table

Description automatically generated

At the size of one hundred, the Linear congruential method produces statistically random output however, this method does not seem to scale well as at larger sizes as by the time 1000 numbers have been generated, there is greater than 99.999% confidence that the output generated is not random. For the sizes, 10k and 10m the possibility of this output being random is so small that it can no longer be displayed easily, even in scientific notation. Another notable fact is that the seed does not significantly impact this generator, as the results were exactly the same with every seed used during testing.

* 1. Lagged Fibonacci chi-squared results:

Table

Description automatically generated

This generator produces varied output based on the seed. Most seeds are still considered random when only 100 numbers are generated; however, at a length of one thousand and above the chi-squared results still show that the numbers are not that random and vary significantly from the expected output.

* 1. Middle-square chi-squared results:

Table

Description automatically generated

The middle-squared method also produces varied output based on the seed. This method shows entropy even at the one thousand level, as after the one hundred level, there is a significant difference in actual output and the expected results.

Since none of these methods can pass the chi-squared test, the next step is to analyze the percent of the possible output range covered. Since the possible range of numbers generated is between 0 and 500 the percent cover for the run of 100 has a maximum of 20% for all other lengths generated 100% coverage is possible. As a disclaimer, it is also possible that the values could be random as at large scales, the actual output is likely to stray from the expected output; however, given the nature of PRNG I do not believe this is the case of my dataset6.

1. Result analysis (percent coverage)
   1. Linear congruential output coverage:

Table

Description automatically generated

For the Linear congruential model, the coverage is only 33% in all seeds. This shows that this method starts looping with only one-third the possible entries covered. This means that this method can't pass the chi-squared test with a possible output range of 500; at smaller output ranges, the Algorithm may perform better.

* 1. Lagged Fibonacci output coverage:

Table

Description automatically generated

In the Lagged Fibonacci model, the seed has an impact on the coverage range; however, on some seeds, it is possible to have over 80% coverage on a range of 500 values. This shows that the chi-squared results are not as heavily impacted by the formula not outputting a wide range of values but rather that some values are prone to come up more frequently due to the algorithm's logic. A smaller output range would still impact the chi-squared results but not as greatly as it would for the linear congruential method.

* 1. Middle square output coverage:

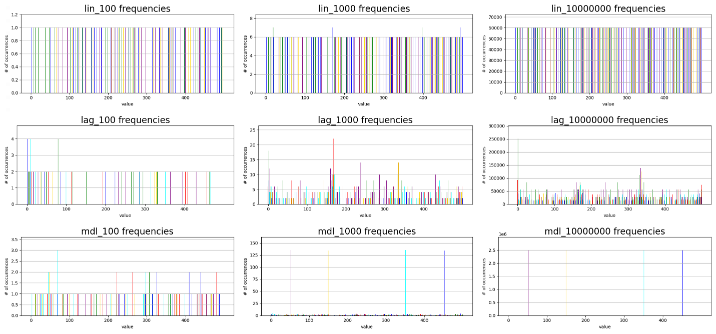
Table

Description automatically generated

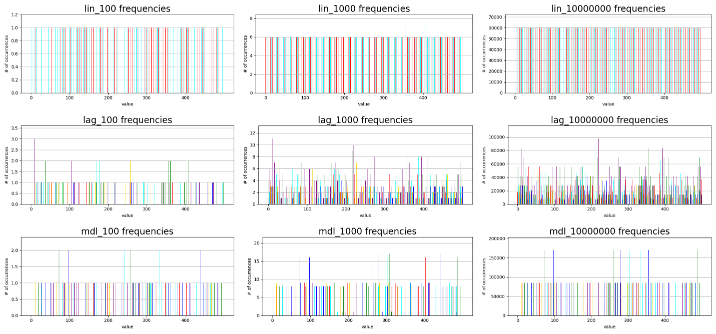
On average, the Middle square method has a smaller range of output than the lagged Fibonacci method but larger than the linear congruential method. Better seeds could see a more extensive range of outputs; however, for the tested seeds the lace of output range would greatly impact the chi-squared test and having a smaller possible range could improve the results of the test.

1. Graphical seed output

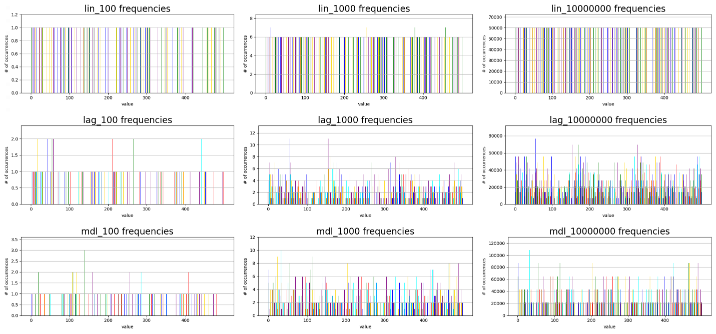
Seed: 11111111



Seed: 26857536



Seed: 39991234



Seed: 57224956

Graphical user interface

Description automatically generated

Looking at the output from each seed visually, it is apparent that the linear congruential method has lower overall output coverage, but the numbers that are outputted are done so consistently appearing about the same number of times. The lagged Fibonacci method has a wider range of outputs however it does favor certain numbers this is ok because true randomness does not say the each number has to occur exactly the same frequency however for the purposes of the chi squared analysis it does show that it is less likely to be random. Finally the Middle square method shows that the Algorithm is prone to entropy with two of the four test seeds heavily favoring a small subset of numbers.

The final point of comparison is the time taken. For the purposes of this test the different algorithms where compared based on the time needed to generate ten million numbers.

Linear congruential is the fastest with the lagged Fibonacci method taking just a bit longer; however, the middle square method takes almost double the time of the other two methods and if the seed results in large values then the time taken seems to grow exponentially as shown in the seed 57224956.

1. Conclusion

While more research is needed on the impact of using a smaller possible output range is needed; for the range of 501 possible outputs the results are as follows. Linear congruential is best used when only a few unique numbers are needed as it has the fastest run time of all tested algorithms but suffers when trying to generate a larger variety of output. The lagged Fibonacci method is best used when large quantities of random numbers need to be generated as it runs faster than middle square method of generating pseudo random numbers. Finally, the Middle square method can be used to generate medium amounts of random numbers. However, this method is prone to entropy and, even on good seeds, it produces a smaller range of numbers than the lagged Fibonacci method. This generator has no real advantage compared to the other methods tested as the implementation is not simpler than the other methods and the results are inferior to the other methods. This method can be improved upon using the XOR technique; however, this process would be more resource-intensive and not a fair comparison to the other methods7.

1. References
2. Vasyltsov, Ihor, Eduard Hambardzumyan, Young-Sik Kim, and Bohdan Karpinskyy. “Fast Digital TRNG Based on Metastable Ring Oscillator.” *Cryptographic Hardware and Embedded Systems – CHES 2008*, n.d., 164–80. https://doi.org/10.1007/978-3-540-85053-3\_11.
3. Avaroğlu, Erdinç, Taner Tuncer, A. Bedri Özer, Burhan Ergen, and Mustafa Türk. “A Novel Chaos-Based Post-Processing for TRNG.” *Nonlinear Dynamics* 81, no. 1-2 (2015): 189–99. https://doi.org/10.1007/s11071-015-1981-9.
4. Belov, A. A., N. N. Kalitkin, and M. A. Tintul. “Unreliability of Available Pseudo-random Number Generators.” *Computational Mathematics and Mathematical Physics* 60, no. 11 (2020): 1747–53. https://doi.org/10.1134/s0965542520110044.
5. *Zulfikar, Hubbul Walidainy. “An Improved Design of Linear Congruential Generator Based on Wordlengths Reduction Technique into FPGA.” International Journal of Electrical and Computer Engineering (IJECE) 5, no. 1 (2015): 55. https://doi.org/10.11591/ijece.v5i1.pp55-63.*
6. *Bernard Widynski. 2017. Middle Square Weyl Sequence RNG. CoRR (2017).*
7. *Berkson, Joseph. "Some difficulties of interpretation encountered in the application of the chi-square test." Journal of the American Statistical Association 33, no. 203 (1938): 526-536.*
8. *Ali-Pacha, Hana, Naima Hadj-Said, Adda Ali-Pacha, Mohamad Afendee Mohamed, and Mustafa Mamat. "Cryptographic adaptation of the middle square generator." International Journal of Electrical & Computer Engineering (2088-8708) 9, no. 6 (2019).*
9. *Sindhikara, Daniel J., Seonah Kim, Arthur F. Voter, and Adrian E. Roitberg. "Bad seeds sprout perilous dynamics: stochastic thermostat induced trajectory synchronization in biomolecules." Journal of Chemical Theory and Computation 5, no. 6 (2009): 1624-1631.*