The trends present within the outputs of Linear Congruential Generators

Research Question: What are the trends of the generated outputs for different implementations of the Linear Congruential Generator algorithm.

Word count: 3487

Subject: Computer Science

Contents

1 Introduction

Computers, the Internet, and software algorithms play a critical role in our lives. Smartphones, cars, entertainment, appliances, and more, rely on layers of complex software algorithms. That software achieves complex results all through basic operations and concepts. One of these concepts which is crucial to modern computing is random numbers. Games involving card shuffling, dice rolling, etc. all rely on randomness. Statistical sampling relies on randomness to pick a representative population [1]. Digital simulations of unpredictable phenomena [2] need enormous amounts of random numbers. And the most important, digital security, depends on random numbers.

Randomness is present all around us, yet computers, cannot create or replicate this randomness. This is because computers operate upon a prescriptive set of instructions. Algorithms called pseudo-random number generators (PRNGS) are used to address this issue by allowing computers to replicate randomness. The first PRNG algorithm created was the Middle Square Method by John von Neumann in 1946 [3]. The creation of new "random numbers" involved taking a large integer and squaring it. The numbers in the middle of the squared integer would be the output. The Middle Square Method often led to repeating sequences of outputs that were not random enough for most applications. Another popular algorithm for generating random outputs was the Lehmer Generator, created by D. H. Lehmer in 1951 [3]. This algorithm outperformed the Middle Square method. The algorithm was later improved by W. E. Thomson A. Rotenberg who created the "Linear Congruential Generator" [3]. This algorithm is still used today for computer-based random number generation. This essay will focus on comparing implementations of this algorithm and identify trends between the parameters used and the randomness of the generated outputs.

2 Background information

2.1 Algorithm

An algorithm as defined by the Merriam-Webster dictionary is "a step-by-step procedure for solving a problem or accomplishing some end" [4]. In the field of computer science, algorithms process and transform numbers and values.

2.2 Random

Randomness is the tendency of something to be unpredictable, something random will have no observable pattern.

2.3 Deterministic

Something deterministic has a determined and fixed output for every input. This is true within computer programs because the steps and operations conducted by a computer are constant and provide the same result every time if the algorithm is given the same input. Random number generators are an example of a deterministic algorithm. The output of an RNG will be the same given the same input. Connecting back to the idea of randomness, a random number generator is not truly random because its outputs can be predicted.

2.4 Entropy

Entropy is defined as the quality of something being uncertain and chaotic, in the context of random number generators, it is the "measure of randomness or uncertainty" [5], and can is typically measured in bits [5]. Computers cannot generate random numbers by themselves, arithmetic may lead to numbers that seem random, but fundamentally, they are not. Entropy is introduced into a computer through user input, sensors, and outside information, one of the most common is system time or the number of nanoseconds for which the computer has been active. Pseudo-random number generators usually have at least one source of entropy which is the seed of the RNG.

2.5 Seed

The seed is an initial value used to "start" a random number generators. Pseudo-random number generators create more values based on an algorithm that modifies this seed. Given the same seed, a pseudo-random number generator will always produce the same outputs. Seeds are typically the only source of entropy that is introduced into a deterministic random number generator. This seed can originate from many different values, ranging from system time to radioactive decay. The seed just needs to come from naturally occurring event that have some sort of random factor attributed

to them [6].

2.6 Java

To test and demonstrate the concepts within this essay, I will be using the Java programming language to execute code, code shown can be easily replicated in other coding languages. Java was selected as the coding language used for code and experiments for this essay because I am most familiar with it, and Java has a native implementation for the random number generator algorithm examined within this paper.

2.7 Modulus

The definition of modulus in a computer science context is the remainder of a division operation. It is also referred to as a "modulo operation", within the code snippets or equations shown within this essay, modulus, or remained will be represented as "%" as that is the way it is the modulus operator within Java.

2.8 Linear Congruential Generator

2.8.1 Basic Overview

A Linear Congruential Generator is a very popular algorithm used to generate random numbers [7]. A Linear Congruential Generator generates a series of numbers based on three constant integers and an initial seed [8]. This algorithm generates "new" numbers through a series of operations. These operations are done using the previously mentioned parameters, the operations modify the original number, in this case, the seed, to generate a new value.

2.8.2 Equation

The equation for a Linear Congruential Generator is defined as:

$$X_{n+1} = (aX_n + c) \,\%\, m \tag{1}$$

"m" is defined as the modulus; $m > 0$

"a" is the multiplier; $0 \leq a < m$

"c" is the addend; $0 \leq c < m$

"X_0" is the starting value; $0 \leq X_0 < m$

2.8.3 Output

The code used in the experiments of this paper outputs a “long” datatype. A “long” is represented on the computer by 64 bits 2^64 worth of data, meaning the output can range anywhere from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807[9]. This output is taken, and modified for the other tests as not all tests require a "long" datatype. The output can be converted to an “integer” data type, which outputs whole numbers from the range of -2147483648 to 2147483647[9], and the “float” data type which outputs from 0 to 1[10], with decimals (Note: float datatype has a larger range however the outputs of this algorithm are intentionally limited to smaller range).

2.8.4 Period

The period of a Linear Congruential Generator is the number of numbers the algorithm generates before X_0 reoccurs as an output, for example:

let $m = 10$, $a = 6$, $c = 6$, $X_0 = 6$:

$$X_{n+1} = (6(6) + 6) \ \% \ 10 = 2$$
$$X_{n+1} = (6(2) + 6) \ \% \ 10 = 8$$
$$X_{n+1} = (6(8) + 6) \ \% \ 10 = 4$$
$$\text{etc.}$$

This sequence beginning with value 6 outputs the following values:

$$6,\ 2,\ 8,\ 4,\ 0,\ 6,\ 2,\ 8,\ 4... \tag{2}$$

The sequence shown has a period of five, meaning it takes five iterations before the pattern repeats. Every single instance of a Linear Congruential Generator, no matter the parameters, will have a finite period because the algorithm is deterministic and will eventually reach the initial number.

2.8.5 Goal of Linear Congruential Generators

Linear Congruential Generators do not produce truly random numbers because the procedures used to generate the next number are constant. Inconsistency can originate from the modification of the constants used in the equation, and most often from the seed "X" which is normally the sole source of entropy for this method of random number generation. As a result, the constants of the equation must be carefully chosen to produce high-quality pseudorandom numbers. Another major goal of Linear Congruential Generators (and other pseudo-random number generators) is to be fast for the computer to execute.

2.8.6 Applications of Linear Congruential Generators

Linear Congruential Generators are deterministic. The parameters of the algorithm can be predicted with relative ease, simply performing opposite operations on the output. Due to this, Linear Congruential Generators perform poorly in the realms of digital security (maybe a source and more explanation). However, they perform exceptionally well at other tasks where the outputs of the algorithm only need to seem random. This is because Linear Congruential Generators are a quick way of creating large quantities of pseudo-random numbers. They can be used in cases where true randomness is not necessary, but large quantities of random numbers are required.

3 Investigation

3.1 Aim

The focus of this investigation will be to compare different common implementations of the Linear Congruential Generator for random number generation. This investigation will compare the outputs of some Linear Congruential Generators to determine the effects of the parameters of the algorithm on its output.

3.2 List of common implementations

This is a list of some commonly used parameters for Linear Congruential Generators. These are real world implementations of this algorithm, mostly in different coding languages [11]. I will be examining these algorithms in my investigation .

Name	Modulus	Multiplier	Increment
Borland C/C++	2^{32}	22695477	1
glibc	2^{31}	1103515245	12345
Turbo Pascal	2^{32}	134775813	1
Microsoft Visual C++	2^{32}	214013	2531011
Microsoft Visual Basic	2^{24}	1140671485	12820163
C++11	$2^{31} - 1$	48271	0
Java	2^{48}	25214903917	11
random0	$2^3 7^5$	8121	28411
cc65	2^{23}	65793	826366247
RANDU	2^{31}	65539	0

Table 1: List of Linear Congruential Generators

3.3 Description of criteria

3.3.1 Runtime speed

The most straightforward criterion is runtime speed which is the amount of time the algorithm takes to generate a new number. This will be tested by taking the system time before and after a random number is generated. To eliminate as potential error, this test will be conducted after every random number, and I will generate 100,000,000 numbers for every single set of parameters.

3.3.2 Period

As described above, the period of a function is how many iterations it takes for the algorithm to repeat. This test will be conducted by storing the initial seed used to initialize the algorithm, and generating new numbers until the seed is generated once more. By counting the number of unique values generated, we can find the period of each set of parameters.

3.3.3 Randomness

The randomness of a Linear Congruential Generator can be quantified in many ways, in this investigation I will be using three different methods.

Range Distribution The first test involves examining the distribution of data across different ranges. Take a six-sided dice, if the die is fair, each side should have a probability of 1/6, same concept applies to larger dice/sources of random numbers. Part of the goal of a PRNG like the Linear Congruential Generator is to emulate true randomness, so the same criteria can be applied here. Each Linear Congruential Generator is essentially a nine-quintillion-sided dice. Now, there are some limitations to this test, firstly, even if you roll a die six times, there is a decent change that you will get a duplicate. To resolve this issue is by rolling that die many times. The more computer-specific problem with this measure of randomness is that nine-quintillion is an absurdly large amount of numbers to generate, additionally, the period of the given parameters may be less than nine-quintillion. To address these issues the amount of numbers will be limited to the a large quantity (100,000,000) and the numbers will be generalized into ranges. This removes the issue of not being able to examine the complete range of theoretical outputs.

Chi-Square test The Chi-Square test is a method to compare observed frequencies versus predicted frequencies in a set of data. This is essentially an extension of the range distribution test, but it does have the ability to quantify the observed difference into a comparable, single value. The setup for this test involves defining two "hypotheses" which will be compared. A hypothesis is a set of frequencies, the first one for the Chi-Square test is the "null hypothesis" which is the expected distribution. If we refer to the example above, rolling a six-sided dice should ideally give us an even spread between all six intervals. The second hypothesis is called the "alternative hypothesis" which is the real-life sample data I will collect. The equation is as follows; for each interval, the alternative hypothesis a is subtracted by the null hypothesis n, squared, divided by n.

$$\Sigma \frac{(a-n)^2}{n} \tag{3}$$

The sum of these deltas is the total score of the algorithm. The result of the Chi-Square test is the "fit" of the data compared to the expected result, this score can be compared to a critical value which is based on degrees of freedom and α [12]. α is an arbitrary number representing a threshold value that determines whether a statistical deviation is significant, in this case, 5% or 0.05. The dF or degrees of freedom value is 9 in all my tests.

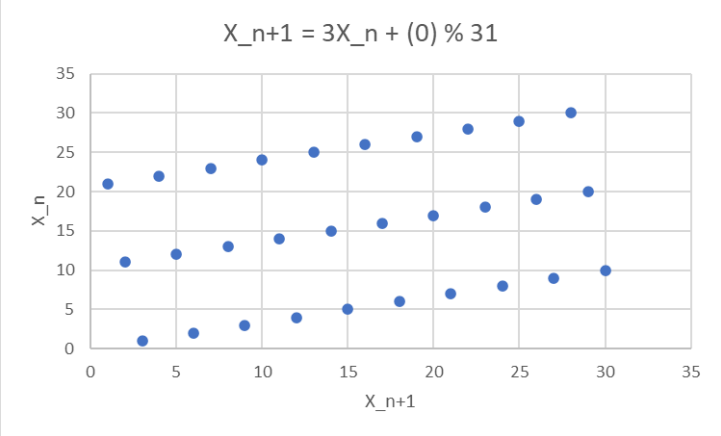
$$dF = (rows - 1)(columns - 1) \tag{4}$$

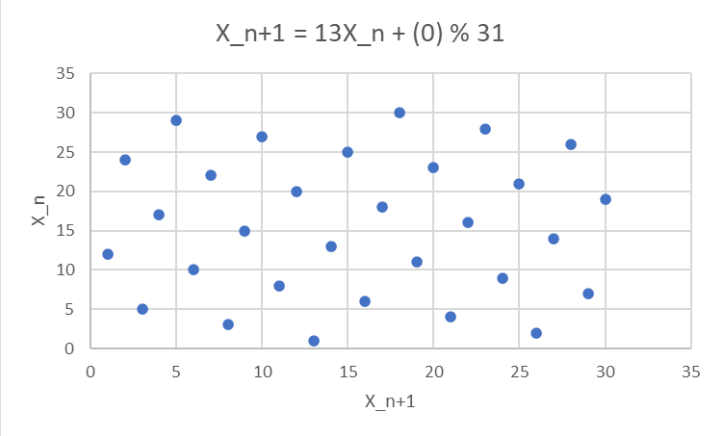
Spectral test The Spectral Test claims to be more effective at predicting the randomness of Linear Congruential Generators [7]. The way this test works is by plotting all the results of one period of an LCG and identifying visual trends. For example, these two Linear Congruential Generators:

$X_{n+1} = (3X_n + (0)) \ \% \ 31$

$X_{n+1} = (13X_n + (0)) \ \% \ 31$

If we plot $(X_n + 1)$ against (X_n), starting with a seed of 1, the distribution of our algorithms looks like this.





The outputs of the LCGs create a linear pattern. These are called hyper-planes, which is how this test determines the "randomness" of a Linear Congruential Generator. In the first image, the graph is split into three distinct hyper-planes. The second graph features seven. We can compare the two algorithms my finding which graph has the least distance between planes which, in this case, is the second graph. When conducting the actual tests there will be a lot more numbers plotted, and it will be hard to notice any hyper-planes. This is solved by plotting the graph in three dimensions, the third dimension being X_{n+2}, hence why the measurement technique is called hyper-planes, and not hyper-lines. The two-dimensional example applies the same concept as the real test, however it is simplified for the sake of explanation.

3.4 Results

3.4.1 Time test results

In order to test how the parameters of each implementation affected the runtime of the Linear Congruential Generator. I used twenty randomly generated seeds and generated 100000000 random numbers. The results for average time across those twenty tests are as follows:

3.4.2 Period test results

Period test results were tested by arbitrarily picking five random seeds which were used for each algorithm, and then the algorithm was run continuously until the output was the same as the original

Name	Average Time (milliseconds)
Borland C/C++	969.3
glibc	972.4
Turbo Pascal	969.9
Microsoft Visual C++	971.3
Microsoft Visual Basic	967.7
Native API	967.3
C++11	968.1
Java	967.5
random0	967.3
cc65	972.5
RANDU	971.7

Table 2: LCG average time test results

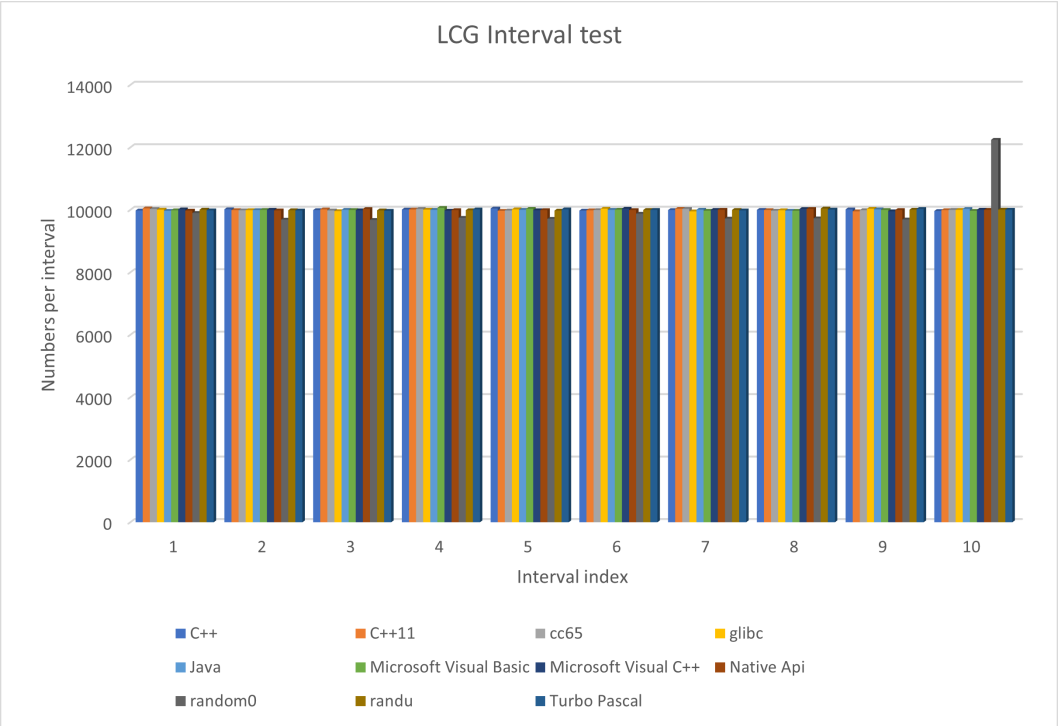
seed.

Name	Average Period (5 tests)
Borland C/C++	4294967296
glibc	2147483648
Turbo Pascal	4294967296
Microsoft Visual C++	4294967296
Microsoft Visual Basic	16777216
Native API	2147483647
C++11	2147483647
Java	281474976710656
random0	134456
cc65	8388608
RANDU	2147483648

Table 3: LCG period test results

3.4.3 Interval test results

To test the distribution of data produced by each Linear Congruential Generator, I outputted 100,000 floating point values between 0.0 and 1.0 which were then plotted along intervals of 0.0-0.1, 0.1-0.2, etc. During my testing, I noticed values would slightly deviate given different seeds so each Linear Congruential Generator was tested using twenty different seeds.



3.4.4 Chi-Square test results

The Chi-Square test used the data set generated by the previous interval test and compared the results against an ideal outcome which would be 100,000 numbers divided evenly among ten intervals. The Chi-Square value for every Linear Congruential Generator is as follows.

Name	Chi-Square value (χ^2)
Borland C/C++	0.485
glibc	0.714
Turbo Pascal	0.319
Microsoft Visual C++	0.532
Microsoft Visual Basic	0.787
Native API	0.263
C++11	0.711
Java	0.282
random0	565.173
cc65	0.434
RANDU	0.297

Table 4: LCG Chi-Square test results

3.4.5 Spectral test results

The spectral test was conducted by plotting three consecutive values generated by the random number generator on a 3d plot (v1, v2, v3; v2, v3, v4; etc.), the plots and raw numbers are available in the appendix. The primary purpose of the test is to compare the distance between planes formed by the outputs of the Linear Congruential Generators. Some of the plots had unique patterns, so I also compared the patterns of different Linear Congruential Generators.

Name	Distance between planes
Borland C/C++	N/A
glibc	N/A
Turbo Pascal	N/A
Microsoft Visual C++	N/A
Microsoft Visual Basic	$7.5E9$
Native API	$1E8$
C++11	N/A
Java	N/A
random0	2222
cc65	4000
RANDU	$7.5E8$

Table 5: LCG Spectral test results

Name	Visible patterns
Borland C/C++	0
glibc	0
Turbo Pascal	0
Microsoft Visual C++	0
Microsoft Visual Basic	14
Native API	324
C++11	0
Java	0
random0	1
cc65	1
RANDU	15

Table 6: LCG Spectral test results

3.5 Evaluation of results

3.5.1 Time test evaluation

From the results of my tests, it can be concluded that the different implementations of the Linear Congruential Generator algorithm had similar runtime, despite different parameters. While there are some differences across the implementations, they can hardly be counted as significant. The difference between the fastest algorithm (random0 and Native API) at $9.673 * 10^{-}9s$ per output, versus the slowest (cc65) at $9.725 * 10^{-9}s$ per output is insignificant. The time notation for the Linear Congruential Generator algorithm is O(n) meaning the time it takes to generate any amount of numbers increases linearly to that amount of numbers, this is independent of the parameters.

3.5.2 Period test evaluation

The results of the period tests dictate that the period of the Linear Congruential Generator did not change based on the initial seed of the algorithm. One pattern that can be discerned from this algorithm is that the period of the algorithm is exactly equal to its modulus parameter. This test concludes that a high modulus values result in a long period which is generally a desirable factor. One possible negative outcome of having a high modulus is that the memory required to store that value is greater. For example, RANDU uses a value of 2^{32} which is worth 32 bits of memory, while Java uses 2^48 which would require 48 bits of memory. On the other hand, the period could be

interpreted as the number of times the Linear Congruential Generator can be reused before the algorithm stops producing unique outputs. This means that the memory cost of a higher period must be considered when choosing parameters for an LCG.

3.5.3 Interval test evaluation

This test reveals that almost all Linear Congruential Generators have a relatively small deviation between each interval of output. On average each set of parameters has a deviation from the expected value of fewer than 100 numbers which is acceptable considering the 10,000 expected value for each interval. The one exception to this rule was the random0 implementation which had significantly greater error. The interval between 0.9 and 1.0 had significantly more numbers than the other interval. One possible reason behind this is the algorithm's unusual modulus which is $2^3 7^5$. Almost all of the algorithms have a modulus that is equal to some power of two (C++11 being the exception, however that is only a power of two minus one), and all the algorithms have a modulus that is significantly higher than random0. In general the parameters of the Linear Congruential Generator slightly affect the distribution of values over ten intervals except in the case of X where it significantly skews values toward upper intervals.

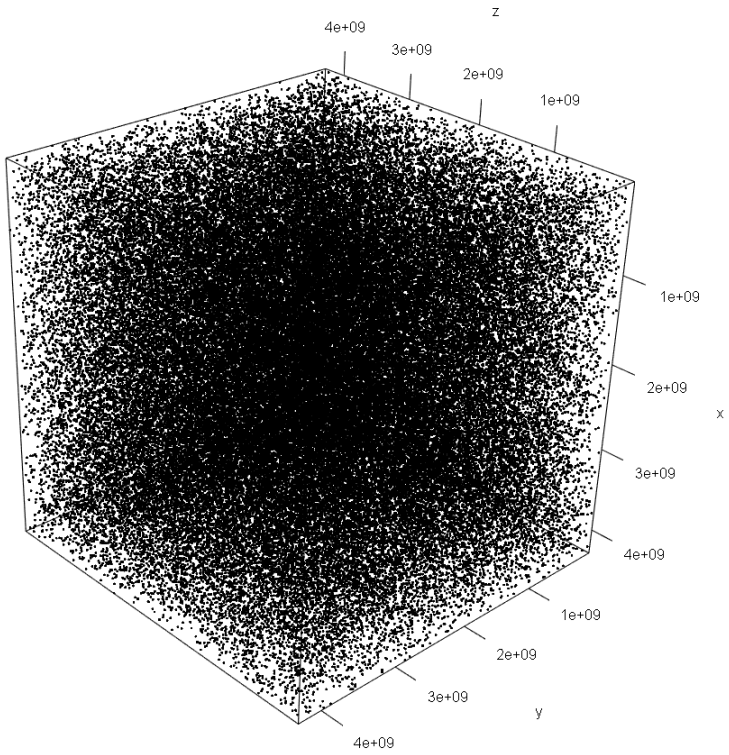
3.5.4 Chi-Square test evaluation

The Chi-Square test is effectively an extension of the previous test and allows us to get a numerical representation of total value deviation across all intervals. As indicated by the previous test, random0 performed exceptionally poorly compared to other algorithms. All other algorithms passed the critical value of 2.26, some with a greater margin. As previously concluded the parameters typically do not have a significant effect on the output of a Linear Congruential Generator across ten intervals.

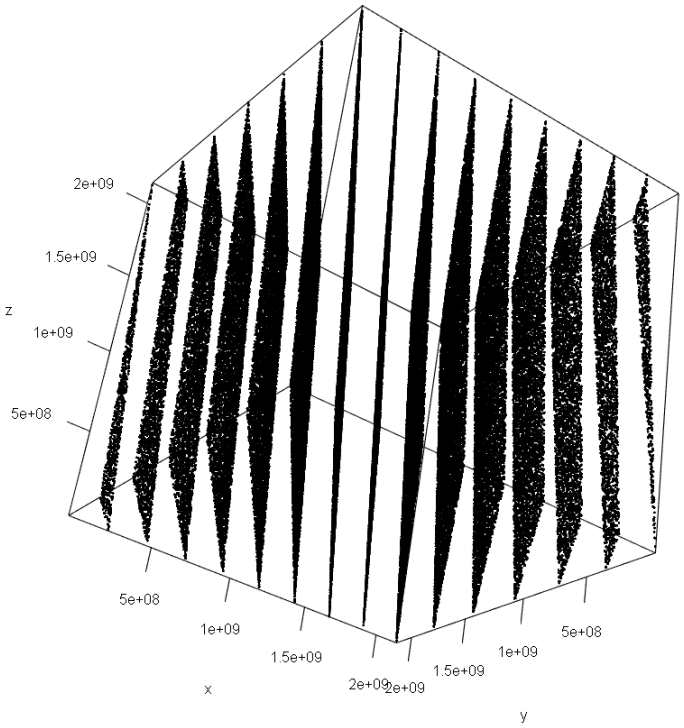
3.5.5 Spectral test evaluation

The results of the spectral test found that some algorithms which had previously passed certain tests had failed this test specifically. Taking the assumption that smaller distances between planes result in a more random output, we can see that Native API, which had previously scored well in the Chi-Square test and time test, scored poorly. This is also reflected in the more subjective,

pattern interpretation section of the spectral test which shows a patterned distribution of data. Note, some algorithms are marked as N/A. This is because some algorithms did not have any noticeable hyper-planes. This limit of visual analysis does still result in some meaningful results, as LCGs with imperceptive planes are effective at generating "random" numbers because the distance between planes is very small.



The C++ algorithm is a good example of a Linear Congruential Generator with no visible patterns. There are limitations to what a still image can represent, however no angle showed visible patterns. An algorithm with a very clear pattern is RANDU



The points plotted are not as randomly distributed as the C++ Linear Congruential Generator. In conclusion, the Spectral test can discern "good" Linear Congruential Generator algorithms from "bad" ones through two methods. The first being distance between planes, which is limited by visual interpretation and subjectively analyzing the patterns. One can assume that Linear Congruential Generators with no pattern are the most random, and generators with a high number of patterns have a broader distribution, and therefore produce more "random" values than Linear Congruential Generators with less visible patterns. This is supported by the more objective distance evaluation as LCGs with no pattern have a small distance between planes.

4 Conclusion

The parameters which I tested in this investigation show the strengths and weaknesses of each individual implementation of the Linear Congruential Generator algorithm. The Linear Congruential

Generator algorithm is not an effective method of creating random numbers because by definition it is only affected by one source of uncertainty and is otherwise deterministic. However it is still possible to evaluate the randomness of data produced by this generator. My tests aimed to evaluate how the parameters used for each Linear Congruential Generator affected the randomness of its outputs. I found that the parameters had a profound effect on the quality of outputs. This exposes the primary weakness of the Linear Congruential Generator algorithm, which is that its viability as a method for generating pseudo-random numbers is dictated by its parameters. My tests showed which implementations of this algorithm produced good and bad outputs. These results could easily be applied when one needs a method of generating random numbers. While a Linear Congruential Generator would not be a suitable source of randomness for something like a password security system, LCGs can be very useful for casual applications. If someone was to make a game that involved rolling dice, a Linear Congruential Generator would be an appropriate choice because Linear Congruential Generators can create seemingly random numbers quickly, however, they must still ensure that they are using appropriate parameters. My investigation revealed the importance of choosing a proper implementation of the Linear Congruential Generator algorithm in order for it to produce desirable pseudo-random number outputs.

The topic of this investigation was to find the effects of a Linear Congruential Generator's parameters on the randomness of its output. I found that the effect was profound and that part of the weakness of the Linear Congruential Generator was its dependency on "good" parameters. These conclusions can be applied to random numbers as concept in computing. Donald Knuth a renowned computer scientist said "Random numbers should not be generated with a method chosen at random"[7]. Randomness is crucial to computing and has many real-world applications and my investigation supports this notion that the generation of random numbers needs to be precise to accurately simulate true sources of randomness.

References

[1] L. Westfall, "Sampling Methods," The Westfall Team, 2008. [Online]. Available: https://www.westfallteam.com/sites/default/files/papers/Sampling%20Methods.pdf. [Accessed: 12-Dec-2022].

[2] M. Shinozuka, "Digital simulation of random processes and its applications," Journal of Sound and Vibration, 25-Jul-2003. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/0022460X72906001. [Accessed: 12-Dec-2022].

[3] P. L'Ecuyer, "History of uniform random number generation," IEEE, 2016. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8247790. [Accessed: 16-Dec-2022].

[4] "Algorithm definition" Merriam-Webster. [Online]. Available: https://www.merriam-webster.com/dictionary/algorithm. [Accessed: 12-Dec-2022].

[5] A. Vassilev and T. A. Hall, "The Importance of Entropy to Information Security," IEEE, 2016. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6756842. [Accessed: 12-Dec-2022].

[6] M. S. Turan, E. Barker, J. Kelsey, K. A. McKay, M. L. Baish, and M. Boyle, "Recommendation for the entropy sources used for random bit generation," Recommendation for the Entropy Sources Used for Random Bit Generation, Jan-2018. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90B.pdf. [Accessed: 13-Dec-2022].

[7] D. E. Knuth, The Art of Computer Programming. Reading: Addison-Wesley, 1997.

[8] K. Sigman, "Random number generators." [Online]. Available: http://www.columbia.edu/~ks20/4106-18-Fall/Simulation-LCG.pdf. [Accessed: 13-Dec-2022].

[9] "Primitive data types" Java Documentation. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html. [Accessed: 12-Dec-2022].

[10] "Random" Java Documentation. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/Random.html. [Accessed: 12-Dec-2022].

[11] "Linear Congruential Generator," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Linear_congruential_generator. [Accessed: 12-Dec-2022].

[12] "Table 1: Critical values (percentiles) for the t distribution.," University of Washington. [Online]. Available: https://faculty.washington.edu/heagerty/Books/Biostatistics/TABLES/t-Tables. [Accessed: 12-Dec-2022].

A Appendix

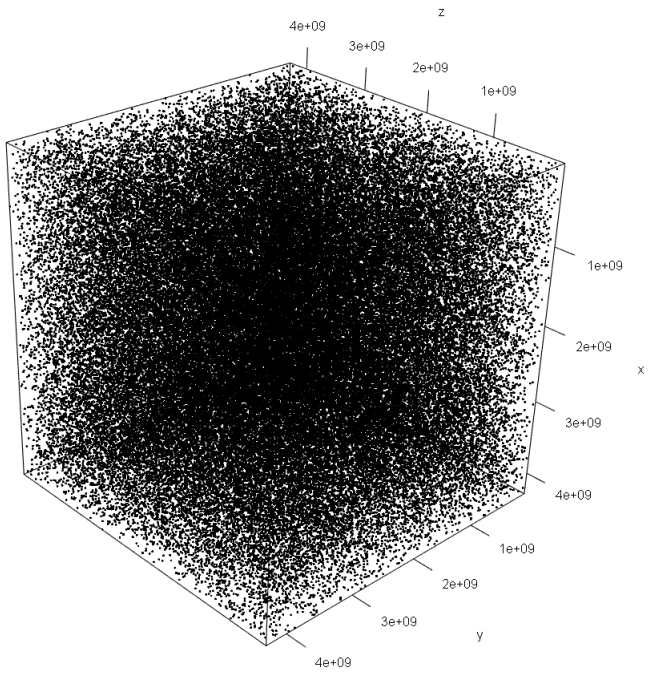
A.1 Code used to perform tests

`https://github.com/x3rmination/ExtendedEssayResources/tree/master/src/main`

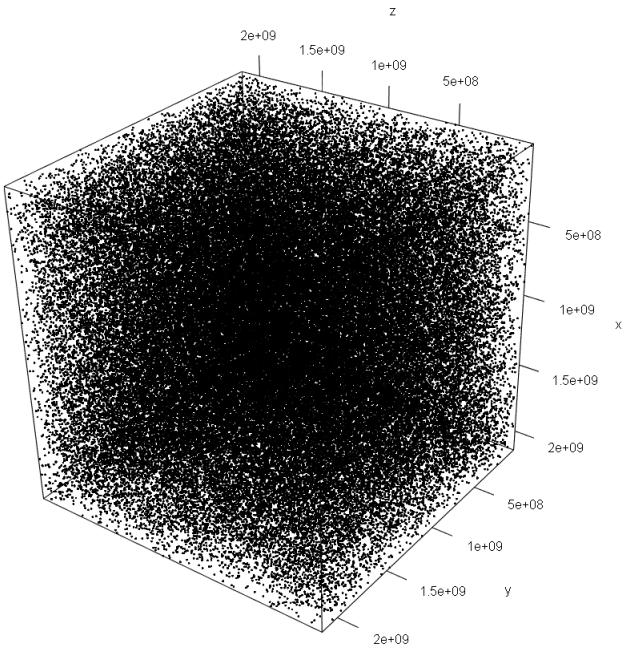
A.2 Results of numerical evaluation tests

C++ C++11 cc65 glibc java microsoftvisualbasic microsoftvisualc++ nativeapi random0 randu turbopascal

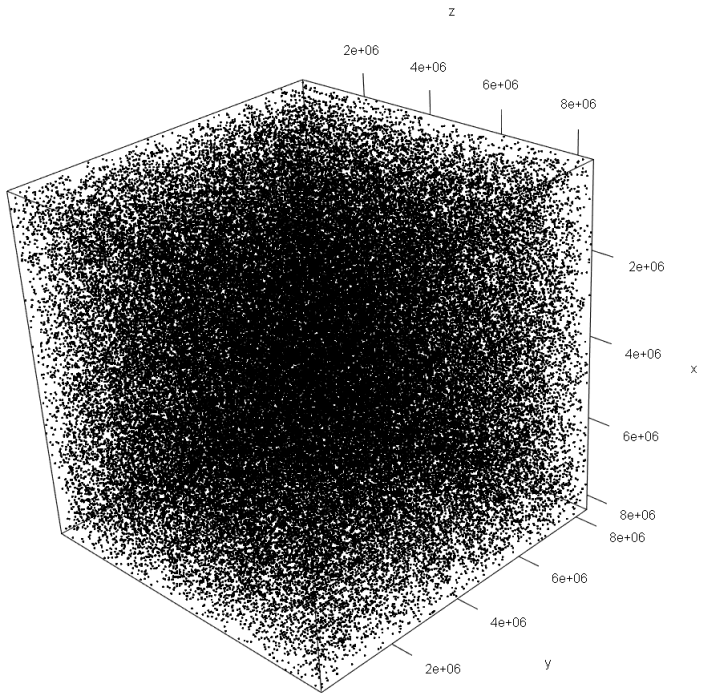
A.3 Results of spectral test



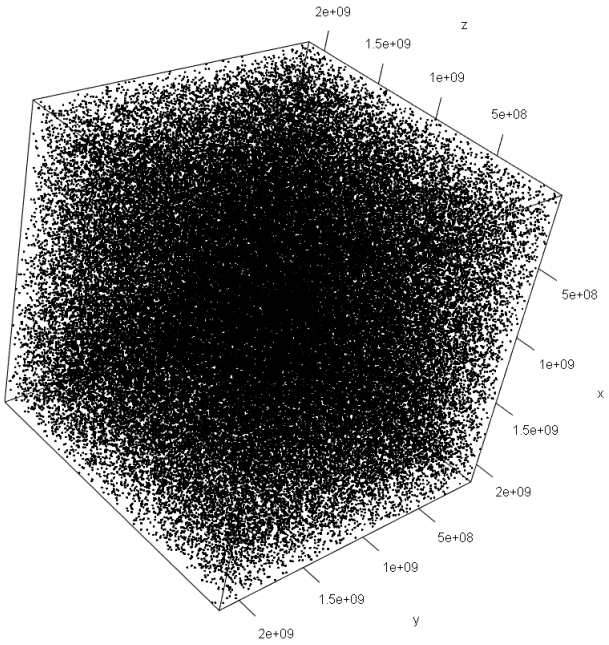
C++


z
2e+09
1.5e+09
1e+09
5e+08
5e+08
1e+09
x
1.5e+09
2e+09
5e+08
1e+09
1.5e+09
y
2e+09

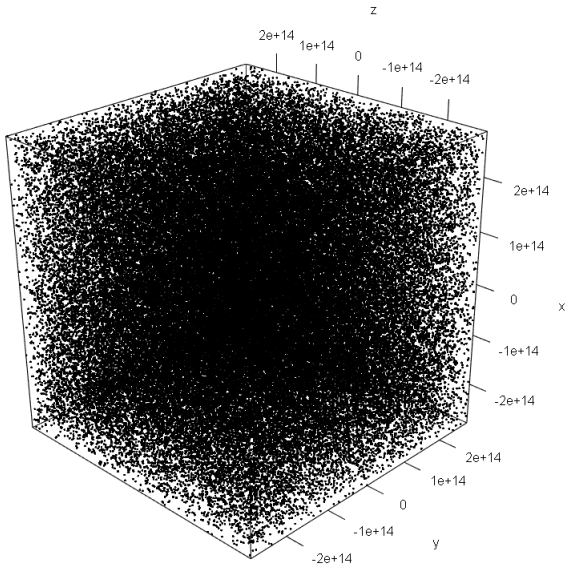
C++11


z
2e+06
4e+06
6e+06
8e+06
2e+06
4e+06
6e+06
8e+06
x
8e+06
6e+06
4e+06
2e+06
y

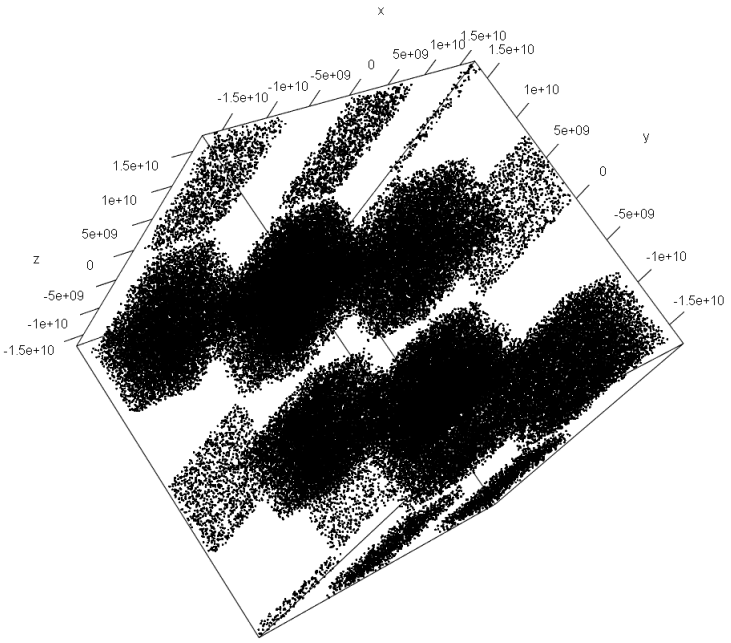
cc65


2e+09
z
1.5e+09
1e+09
5e+08
5e+08
1e+09
x
1.5e+09
2e+09
5e+08
1e+09
1.5e+09
y
2e+09

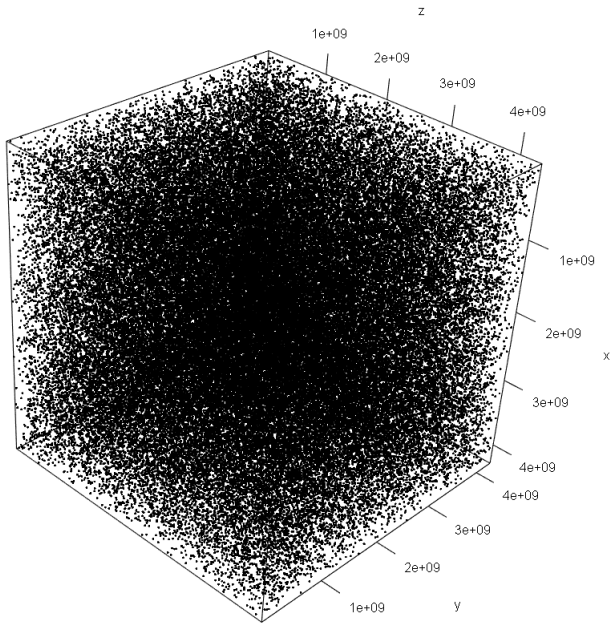
glibc


z
2e+14
1e+14
0
-1e+14
-2e+14
2e+14
1e+14
0
x
-1e+14
-2e+14
2e+14
1e+14
0
-1e+14
y
-2e+14

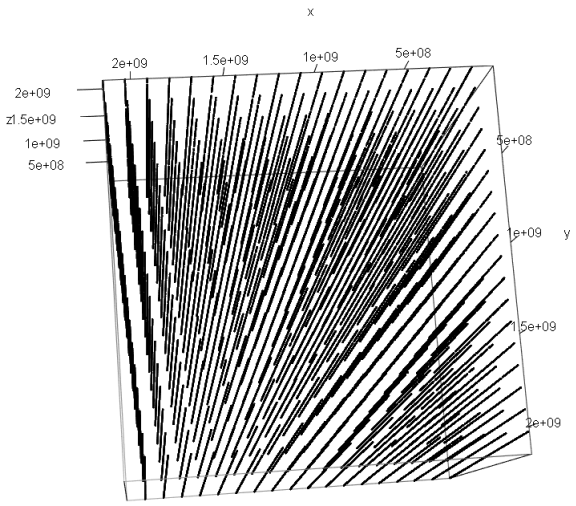
Java



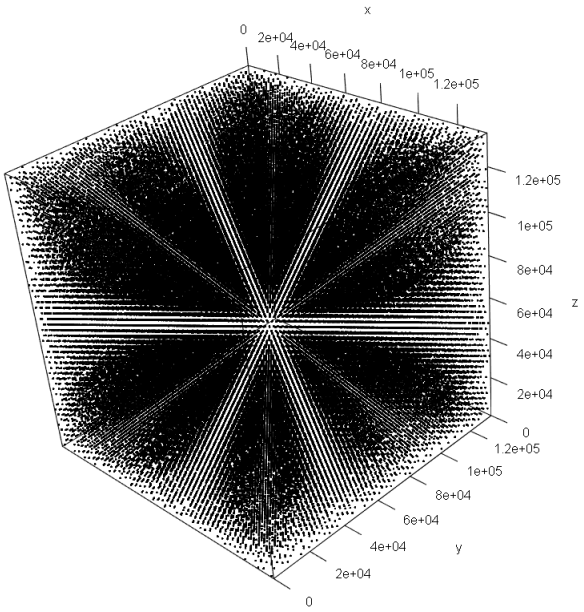
microsoftvisualbasic



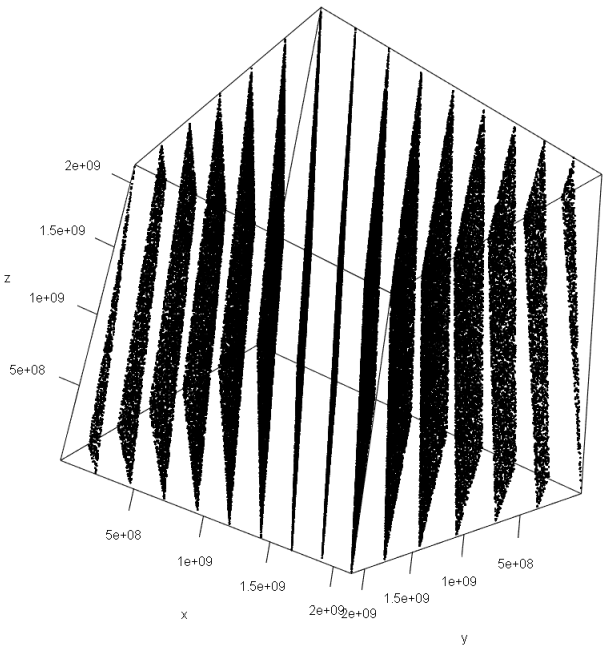
microsoftvisualc++


x
2e+09
1.5e+09
1e+09
5e+08
2e+09
z1.5e+09
1e+09
5e+08
5e+08
1e+09
y
1.5e+09
2e+09

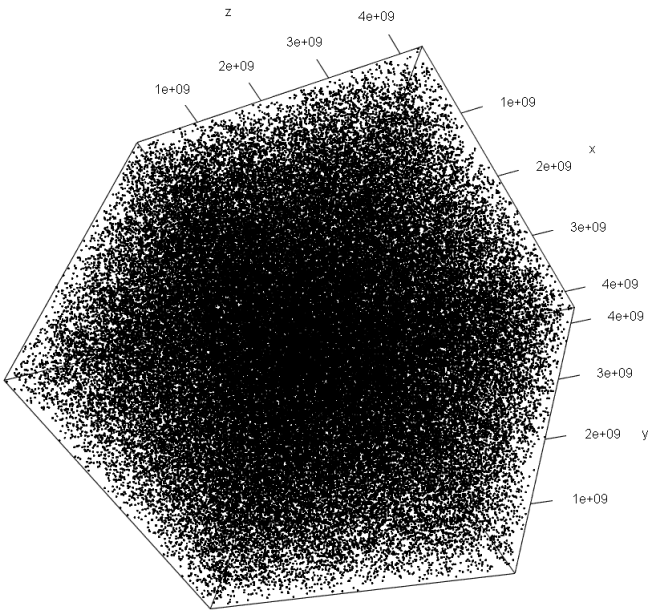
nativeapi


x
0
2e+04
4e+04
6e+04
8e+04
1e+05
1.2e+05
1.2e+05
1e+05
8e+04
6e+04
4e+04
2e+04
0
z
1.2e+05
1e+05
8e+04
6e+04
4e+04
y
2e+04
0

random0


2e+09
1.5e+09
z
1e+09
5e+08
5e+08
1e+09
1.5e+09
2e+09
x
2e+09
1.5e+09
1e+09
5e+08
y

randu



turbopascal