An Investigation Into the Quality of Different Types of Random Number Generators

Which type of random number generator most consistently produces sufficiently random binary sequences?

Computer Science Extended Essay

Examination Session: May 2021

Word Count: 3843

Contents

Introduction

Randomness is the "[t]he quality or state of lacking a pattern or principle of organization; unpredictability". For example, when rolling a die, the outcome of any particular roll is unpredictable. Randomness has always been of interest to philosophers and mathematicians, but interest in random numbers specifically grew after John Venn published the 1888 edition of his book Logic of Chance.

In Logic of Chance, Venn attempted to visualize randomness by building a "random path" by assigning a direction to each number (north to 0, north-east to 1, etc. except for 8 and 9), and drawing a line for each number in the given direction. The number Venn chose to illustrate was π, as it was considered random at the time (though we now know it not to be).[1]

In 1955 the RAND Corporation published a document entitled A Million Random Digits with 100,000 Normal Deviates. This document was developed by rerandomizing a table of digits generated by the random frequency pulses of an electronic roulette wheel. RAND stated that "the purpose of producing such large tables was to meet the growing need for random numbers in solving problems by experimental probability procedures". Despite this seemingly huge supply of random numbers, it was just the beginning of mass production and availability of random digits.[2]

Due to the need for a much larger supply of random numbers, arithmetic procedures, or algorithms, were developed to generate huge amounts of random numbers. One of the first methods was the Middle-Square Method which was devised by John von Neumann.[3] The Middle-Square Method generates groups of n random digits by starting with a number n digits long, squaring it, and taking the middle n or n + 1 digits for the next squaring and so on. However, this method was discovered to be terrible source of random numbers. The issue arises due to how numbers square, for example squaring the number 3792 results in 14,379,264, which with the middle digits 3792 it will result in the exact same number. Methods like this are no longer used, and modern methods are discussed in Section 3.

Random numbers are increasingly important in the digital age as their use cases increase in number. Now, random numbers are used in applications from games, lotteries, and simulation, to critical uses in cryptography and due to random data's critical use in encryption of digital data, the generator that outputted the data must be truly random so that the encrypted data is secure. This study aims to determine what random number generator most consistently generates sufficiently random data. In this context, random data is defined as a binary string of 0s and 1s.

To test this, 500 1000 digit binary strings were generated from a variety of popular random

1. John Venn, *The Logic of Chance* (Dover Publications, 2013).
2. Deborah J Bennett, *Randomness - Deborah J. Bennett*, https://www.hup.harvard.edu/catalog.php?isbn=9780674107465.
3. Enacademic, *Middle-square method*, https://enacademic.com/dic.nsf/enwiki/393440.

number generators and were then ran through a statistical test suite designed by the National Institute of Standards and Technology (NIST), of which results were printed and plotted on a graph for analysis of the results.

1 Definitions

Word	Definition
Bit	A single binary digit, either a 0 or a 1
P-Value	A value between 0.0 and 1.0, which serves as evidence against a null hypothesis, with the hypothesis being that the data is random in this case. An acceptable P-Value is ≥ 0.01 for the purposes of this research (unless otherwise noted).
n	Length of a given binary string
ε	A given binary sequence
S_{obs}	Absolute value of the sum of a sequence
M	Length of each block of binary digits from random binary string
$\chi^2(obs)$	A measure of how well the observed proportion of ones within a given M-bit block match the expected proportion ($\frac{1}{2}$)
$V_n(obs)$	The total number of runs (i.e., the total number of zero runs + the total number of one-runs) across all n bits
$igamc$	Incomplete Gamma Function
$erfc$	Complementary Error Function

2 What is randomness?

Humans naturally have a sense of randomness, for example, if we flipped a fair coin (P('heads') = $\frac{1}{2}$) 20 times twice and got the sequences:

HTHTHTHTHTHTHTHTHTHT

HTHTTHHHTHTHHTTTTHTH

Most people would describe the second one as random and the first as not, even though both sequences have the same probability of occurring ($\frac{1}{2}^{20}$).

The "sense" that comes with random events and sequences, makes it difficult to quantify mathematically. This is called subjective randomness,[4] which implies that randomness is not present in the real, objective world, but rather only in the mind of the thinking human. Other definitions are defined as being unable to predict the next number. Take the sequence 1, 2, 3, 4, 5... for example, most people would think that the next number in the sequence is 6, but if someone can't predict that – does that make the sequence sufficiently random?

Another way to look at a random sequence is how complex it is. This concept is called Kolmogorov complexity,[5] named after Andrey Kolmogorov, and defines complexity as the length of a computer program or sentence needed to describe and reproduce a given object. The more complex the description is, the more likely it is to be random. Taking the sequences from the coin flips earlier, how could we describe them? The first could be "HT x 10", which is very short and not complex, but the second has no description other than the sequence itself, "HTHTTHHHTHTHHTTTTHTH", making it complex and random. Most strings of a length n do not have a description shorter than n, and listing the string itself is the best possible description in these cases.

But what is randomness? Randomness is the state of a sequence meeting the above definitions of randomness: feeling random, being nondeterministic, and hard to describe. Based on these definitions it may seem difficult to quantify the "randomness" of a sequence, and it very much so is for "feeling" randomness, but Section 4 of this paper outlines some of many statistical calculations used to determine how nondeterministic a sequence is and determine if it truly is random.

4. Griffiths TL;Daniels D;Austerweil JL;Tenenbaum JB; *Subjective randomness as statistical inference*, https://pubmed.ncbi.nlm.nih.gov/29524679/.

5. Lance Fortnow, *Kolmogorov Complexity*, https://people.cs.uchicago.edu/~fortnow/papers/kaikoura.pdf.

2.1 How do computers generate random numbers?

Based on the definition of randomness we have established, a sequence feeling random, being nondeterministic, and hard to describe, it sounds like with computers, very deterministic and predictable machines, it would be difficult to generate randomness. However, it turns out that there are a variety of methods that computers can use in order to generate randomness. Computers use what is called a Random Number Generator (RNG) to generate sequences of numbers that can be considered random, of which there are two types: True (or Hardware) Random Number Generators and Pseudorandom Number Generators. These generators require the use one of many generation methods in order to generate a random sequence.

2.2 Pseudorandom Number Generators (PRNG)

A pseudorandom number generator is an implementation of an algorithm to generate a sequence of numbers whose properties approximate those properties of sequences of random numbers.[6] A clear issue is made apparent in both the last sentence and the name of a PRNGs, they are not truly random, they are pseudorandom. This is because a PRNG's output is initialized by an initial value called a seed. This seed may be or partially be truly random values, but knowledge of the seed means that anyone with access to the algorithm can reproduce the same output, meaning it is deterministic.

This deterministic nature leads to potential problems with using PRNGs to get random numbers. PRNGs are "periodic", meaning that eventually they will produce same sequence, which is very undesirable.[7][8] Even though this seems like a large issue that should be handled, most modern PRNGs have periods so large that it is inconsequential. Due to this, it may sound like PNRGs are totally unfit for applications that require secure, nondeterministic numbers, and that is mostly true. However, special cryptographically secure PNRGs (CSPNRG) have been developed but are out of scope in this paper.

One popular algorithm used by PRNGs to generate random numbers is the Linear Congruential Generator (LCG), which yields a sequence of pseudorandom numbers calculated with a discontinuous piecewise linear equation. This algorithm is explained further in section 3.1.

6. NIST, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, April 2010, 14, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf.

7. Mads Haahr, *True Random Number Service*, https://www.random.org/randomness/.

8. Wolfram, *Wolfram Demonstrations Project*, https://demonstrations.wolfram.com/LinearCongruentialGenerators/.

2.3 True(Hardware)Random Number Generators (TRNG)

A true random number generator uses data from truly random, nondeterministic natural events (called entropy) and introduce it to a computer which recognizes slight, unpredictable variations in the data which it then digitizes them into binary data, which the computer can understand and use.[9] Because TRNGs do not utilize a "seed" to initialize some process to derive a number, rather they use a direct, nondeterministic source to derive a sequence of numbers. This means that by nature they meet all the properties of sequences of random numbers, and the sequence will not be reproduced.

Many natural events meet the nondeterministic requirement to be used as a source of entropy in a TRNG. Some of the most popular implementations are RANDOM.ORG's use of atmospheric noise,[10] HotBits's use of measuring radioactive decay,[11] Lavarand's use of lava lamps,[12] and The Australian National University's use of quantum fluctuations in a vacuum.[13] Due to the outputted sequence being truly random, TRNGs are used for a variety of secure applications including cryptography.

A simple TRNG might have a process similar to the one that follows.[14] The entropy is measured by the computer and is digitized for further analysis, then is put through a post-processing to randomize the data further (such as XORing the output of 2 TRNGs), then that output is ran through a series of statistical tests to ensure that the data is still random. Depending on the application, this truly random data can be then used to seed a CSPRNG, which can generate more numbers faster than a typical TRNG.

9. NIST, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, 14.
10. Haahr, *True Random Number Service*.
11. John Walker, *HotBits:* https://www.fourmilab.ch/hotbits/.
12. Inc. Haggerty Enterprises, https://web.archive.org/web/19971210213248/http://lavarand.sgi.com/.
13. Australian National University, *Quantum random numbers*, https://qrng.anu.edu.au/.
14. Watchdog, *Hardware Random Number Generators*, https://cerberus-laboratories.com/blog/random_number_generators/.

3 Sources of Randomness

3.1 Linear Congruential Generator

The Linear Congruential Generator (LCG) is an PRNG algorithm for generating sequences of pseudorandom numbers.[15] Linear Congruential Generators are one of the oldest and most well-known algorithms for generating pseudorandom numbers due to their ease of implementation and need for little memory.[16] Its function is defined as

$$X_{n+1} = (aX_n + c) mod\ m$$

Where X is the sequence of pseudorandom values,
$m, 0 < m$ is the "modulus",
$a, 0 < a < m$ is the "multiplier",
$a, 0 \leq c < m$ is the "incriment",
$X_0, 0 \leq X_0 < m$ is the seed.[17]

The problem with Linear Congruential Generators (and all pseudorandom number generators), is that due to their seeded nature, eventually a pattern will emerge, and numbers will repeat. This is caused by both a weak input seed, as well as weak parameters for the multiplier, increment, and modulus. It is important that the parameters for a specific implementation are tested so that they produce sufficiently random numbers and do not produce an obvious pattern.

3.2 RANDOM.ORG

RANDOM.ORG generates true random numbers using a TRNG, and their natural randomness source of choice is atmospheric noise.[18] Their setup for generation includes a distributed series of three nodes throughout the world, with each node containing a series of three individual radios (sources) feeding into sound cards on PCs running Debian Linux and a series of custom software. Each source generates 12,000 bits a second for a total of 108,000 bits per second generated, this seems like a lot, but it is all very necessary due to the process that the bits go through during generation.[19]

15. Aaron Schlegel, *Linear Congruential Generator for Pseudo-random Number Generation with R*, January 2018, https://aaronschlegel.me/linear-congruential-generator-r.html.
16. Aaron Schlegel, *Permalink to Linear Congruential Generator for Pseudo-random Number Generation with R*, https://aaronschlegel.me/linear-congruential-generator-r.html.
17. Cornell, http://pi.math.cornell.edu/~mec/Winter2009/Luo/Linear%20Congruential%20Generator/linear%20congruential%20gen1.html.
18. Mads Haahr, *True Random Number Service (History)*, https://www.random.org/history/.
19. Mads Haahr, *True Random Number Service*, https://www.random.org/statistics/source-purity/.

As part of the generation process, the bits go through a distilling process to "purify" the data. The data is tested to see if it is skewed towards either zeros or ones, and before generation it is important to adjust and fix this skew. RANDOM.ORG uses a skew correction algorithm developed by John von Neumann, which works as follows: it splits the raw source into pairs, and they are checked, if both bits are the same (00 or 11), both bits are discarded. If the two bits are different, the first bit is added to a new stream of processed bits. Even though this produces an exceptionally good stream of true randomness, running the algorithm on the original 108,000-bit stream results in a loss of nearly 3/4 of the raw data.[20]

3.3 HotBits

HotBits generates true random numbers using a TRNG, and their natural randomness source of choice is the beta decay of Cæsium-137.[21] Despite knowing so much about radiation and chemical decay, it turns out that despite knowing that an element will decay, nature does not tell us when it will, meaning it can be used as a source of true randomness. The hardware setup for HotBits includes a commercial radiation monitor order to measure the decay, HotBits uses a commercial radiation monitor and a computer running Fedora Core Linux and the custom generation software. The setup generates 1600 bits per second, and HotBits runs two identical generators for hardware redundancy purposes.[22][23]

The custom generation software works based on the fact that we can never truly know when a given Cæsium-137 nucleus will decay (meaning its random), the interval between a pair consecutive decays is also random.[24] The software measures a pair of intervals and outputs a zero or a one based on the relative length. However, if the interval is the same the measurement is discarded and neither a zero nor a one is outputted. HotBits does not use a purification method akin to RANDOM.ORG, however the interval comparison for successive bits is reversed to remove the very minimal (order of $10 * 10^{14}$) bias due to radioactive decay. John Walker, HotBits' creator, says "[o]ne of the major advantages of a radioactive source is that it does not require [purification] to remove bias such as that introduced by duty-cycle inequality in sources that use a noise diode to latch the output of a fast multivibrator".[25]

3.4 Australian National University's Quantum Random Number Generator

The Australian National University developed a method to generate true random numbers by measuring quantum fluctuations in their vacuum. Traditionally, a vacuum is a space that

20. July 2020.
21. Walker, *HotBits:*
22. John Walker, *HotBits Driver Program*, https://www.fourmilab.ch/hotbits/source/hotbits-c3.html.
23. July 2020.
24. John Walker, *How HotBits Works*, https://www.fourmilab.ch/hotbits/how3.html.
25. .

is empty of matter or photons. In quantum mechanics, however, that same space resembles a sea of virtual particles appearing and disappearing all the time. This means that the electromagnetic field of the vacuum exhibits random fluctuations in phase and amplitude.[26] By measuring these fluctuations, they are able to generate high-bandwidth random numbers.

By nature, quantum mechanics are random, which gives the Australian National University's group confidence that their numbers are truly random. "We are measuring the quantum fluctuations of the vacuum and quantum mechanics predicts that it will be random. If we can find some patterns in the numbers then physicist would have to work on a better theory to explain what is going on."[27] Despite this confidence, they use a purification method to remove any outside influence on their random numbers. A custom method is used to isolate outside sources of noise that may influence the quantum fluctuations, such as electronic noise.[28]

26. University, *Quantum random numbers*.
27. Australian National University, *Frequently asked questions*, https://qrng.anu.edu.au/FAQ.php.
28. University.

4 Methods of Testing

In April 2010, the NIST released a document entitled A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications, which outlines their suite of statistical tests for testing random and pseudo random number generators. Given a binary string, each test will output a P-Value which can be compared to the ideal P-value of 0.01, and if the outputted P-Value is greater than or equal to, the sequence can be defined as random.[29] For the purposes of this research, the P-Values are analyzed in 2 different ways, using the Proportion of Sequences Passing a Test and the Uniformity of P-Value Distribution test. The following sections provide an overview of the tests used in this research, as well as an example of their application.

4.1 Frequency (Monobit) Test

The Frequency (Monobit) Test[30] is ran with the aim of testing the proportion of zeros and ones in a given sequence to determine if the number of zeros and ones are approximately the same as what would be expected of a truly random sequence. The test assesses the closeness of the fraction of ones to 1/2, meaning that the number of zeros and ones should be about equal.

The math for the test is as follows:

1. The zeros and ones of the inputted sequence are converted to values of -1 and +1 respectively and are added together to produce
$$S_n = X_1 + X_2 + \ldots + X_n$$
2. The test statistic S_{obs} is calculated with the formula
$$S_{obs} = \frac{|S_n|}{\sqrt{n}}$$
3. The P-Value is then calculated with the formula
$$P - Value = erfc(\frac{S_{obs}}{\sqrt{2}})$$

4.2 Frequency Within a Block Test

The Frequency Within a Block Test[31] is ran with the purpose of testing the proportion of ones within blocks of M-bit lengths, to determine if the frequency of ones in the block is

29. NIST, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, 15-16.
30. NIST, 24-25.
31. NIST, 26-27.

approximately 1/2, as would be under the assumption of randomness. While this sounds similar to the first test (Frequency (Monobit) Test), while an entire sequence may be random, that sequence may be split up into smaller ones, and those smaller sequences (blocks) may not be as sufficiently random as the whole sequence.

The math for the test is as follows:

1. The input sequence is split into $N = \frac{n}{m}$ non-overlapping blocks. If there are leftover bits, discard them so that all blocks are evenly sized.
2. Determine the proportion of π_i of ones in each M-bit block with the equation:

$$\pi_i = \frac{\sum_{j=1}^{M} \varepsilon(i-1)M + j}{M}$$

3. Compute the χ^2 value with:

$$\chi^2(obs) = 4M \sum_{i=1}^{N} (\pi_i - \frac{1}{2})^2$$

4. Calculate the P-Value with the formula

$$PValue = igamc(\frac{N}{2}, \frac{\chi^2(obs)}{2})$$

4.3 Runs Test

The Runs Test[32] is ran with the purpose of finding the total number of runs in the sequence, where a run is an uninterrupted sequence of identical bits, to find out if the number of runs of ones and zeros of various lengths are as expected for a random sequence. If the oscillation of ones and zeros is too fast or too slow, the P-Value will reflect that.

The math for the test is as follows:

1. Calculate the pre-test proportion of ones in the input sequence

$$\pi = \frac{\sum j \varepsilon j}{n}$$

2. Determine if the frequency test is passed with $|\pi - \frac{1}{2}| \geq \tau$ where $\tau = \frac{2}{\sqrt{n}}$, and if this equation is true, the runs test is applicable

32. NIST, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, 27-29.

3. Compute the test statistic

$$V_n(obs) = \sum_{k=1}^{n-1} r(k) + 1$$

where $r(k) = 0$ if $\varepsilon_k = \varepsilon_{k+1}$ and $r(k) = 1$ otherwise

4. Calculate the P-Value with

$$P - Value = erfc(\frac{|V_n(obs) - 2n\pi(1-\pi)}{2\sqrt{2n\pi}(1-\pi)})$$

4.4 Proportion of Sequences Passing a Test

Given the empirical results for a statistical test, you can compute the proportion of sequences that pass the given test. This is called the Proportion of Sequences Passing a Test[33] test. The range of acceptable proportions is determined using the confidence interval defined as

$$\rho \pm 3\sqrt{\frac{\rho(1-\rho)}{m}}$$

where $p = 1 - a$ and m is the sample size. If the data falls out of this interval, there is evidence that the data is non-random. In order to get accurate results, the sample size must be sufficiently large ($n \geq 1000$).

4.5 Uniform Distribution of P-Values

The Uniform Distribution of P-Values[34] tests the uniformity of P-Values, as plotted on a graph (histogram) where the interval between 0 and 1 is divided into 10 sub-intervals and the P-Values that lie within each sub-interval are counted and displayed. It is calculated with a combination of formulas, first a chi-squared test is performed on the data with the formula,

$$\chi^2 = \sum_{i=1}^{10} \frac{(F_i - \frac{s}{10})^2}{\frac{s}{10}}$$

where F_i is the number of P-values in sub-interval I and s is the sample size. Then the output of which is used to calculate the P-Value using

$$P - Value = igamc(\frac{9}{2}, \frac{\chi^2}{2})$$

If the P-Value is ≥ 0.0001, the P-Values can be considered uniformly distributed. In order to get accurate results, at least 55 sequences must be processed.

33. NIST, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, 90.
34. NIST, 91.

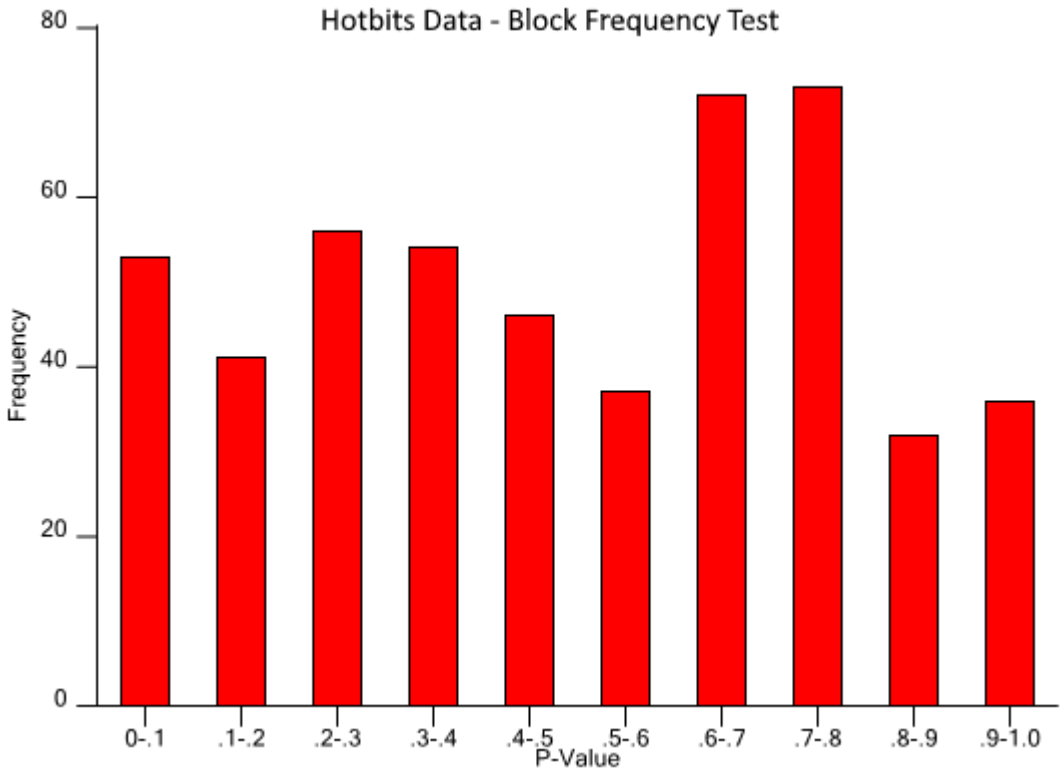
5 Data Collection

In order to collect the data required for this analysis, an implementation of a LCG (using the most popular default parameters) was used to generate 500,000 binary digits which was split into 500 1,000 bit lines in a text file. The remaining sources (RANDOM.ORG, HotBits, and the Australian National University's QRNG) have public APIs (Application Programming Interface) that can be used to download specified amounts of bits, so I sent 500 requests for 1,000 bits each and similarly wrote them to a text file to be ran through a custom program to analyze the data.

6 Results

In order to perform the statistical test suite, I developed a program in Rust that takes an input file of 500 1000 bit randomly generated binary strings, which would produce all the data and graphs necessary to assess the random number generators that are examined in this research. Running the analysis program on a given data set will result in a full debug output of all calculated values, as well as a graph showing the distribution of P-Value frequencies between 0.0 and 1.0. One of the outputted graphs can be seen below, however the remaining graphs can be found in Appendix D.

Figure 1: *Results of Block Frequency Test on HotBits data:*
Real Proportion P-Value: 0.984 ≥ Theoretical Proportion P-Value: 0.9899406 Expected amount of sequences had an acceptable P-Value: (True) Uniform Distribution P-value = 0.72995234 The data is considered uniformly distributed: (True)



7 Testing Conclusions

A multitude of factors must be considered when determining the quality of a certain random number generator. For the purposes of this research the factors considered will be if there are any duplicates in a data set, the number of tests failed, the Proportion of Sequences Passing a Test, as well as the Uniformity of the P-Value Distribution for a given test.

One of the most basic checks to perform on a set of random data is to check for any duplicate strings, as with $1000^2 = 1,000,000$ possible combinations and a data set of only 500 strings there should be no duplicates in our data. Interestingly, the only generator to output a string that occurred multiple times was the HotBits generator with a string that occurred 4 times. This was unexpected because the LCG PRNG would be the expected generator to create duplicate strings due to its periodic nature. This isn't a major concern for someone looking to use random data because as long as the duplicates are generated far enough apart it will be unnoticed. The failed tests are listed in the table below:

Random Number Generator	Statistical Test	# of Failed Tests out of 500
Hotbits	Block Frequency Test	8 (1.6%)
Hotbits	Frequency Monobit Test	12 (2.4%)
Hotbits	Runs Test	7 (1.4%)
LCG	Block Frequency Test	5 (1%)
LCG	Frequency Monobit Test	9 (1.8%)
LCG	Runs Test	4 (0.8%)
RANDOM.ORG	Block Frequency Test	4 (0.8%)
RANDOM.ORG	Frequency Monobit Test	4 (0.8%)
RANDOM.ORG	Runs Test	7 (1.4%)
Quantum RNG	Block Frequency Test	3 (0.6%)
Quantum RNG	Frequency Monobit Test	11 (2.2%)
Quantum RNG	Runs Test	6 (1.6%)

None of these failed test totals are overly concerning because they are a very small percentage of the total 500 binary strings per data set. Ideally random data would be tested for quality through a variety of tests (similar to the ones used here) to see if they're sufficiently random before use to ensure data that would fail like this is not used, meaning it is even less of a concern.

Using the number of failed tests one can determine the Proportion of Sequences Passing a Test. Every test had a confidence interval, or expected proportion of tests passing of 0.9899406, meaning that as long as the proportion of tests passing is greater than that value then a sufficient proportion of tests are passing, which every data set did. The final test applied is the Uniformity of the P-Value Distribution for a given test. Every P-Value distribution for a given data set and test produced an acceptable P-Value, meaning the distribution of P-Values is acceptable, and the P-Values can thus be considered evenly distributed. This is important because the data generated from each source should be a variety of random

data, with various P-Values occurring, so that it is not skewed.

8 Conclusions

Every generator's outputted data passed every statistical test, but which type of random number generator most consistently produces sufficiently random binary sequences? In order to determine this, all the factors discussed in Section 7 Testing Conclusions must be considered. Based on the results there are two main candidates to answer the presented question, the LCG and RANDOM.ORG. Both had no duplicate binary sequences, very few failed tests which resulted in passing of the Proportion of Sequences Passing a Test test, as well as acceptable P-Values from the Uniformity of the P-Value Distribution test. RANDOM.ORG seems to be the generator that most consistently produces sufficiently random binary sequences, due to the fact that it passed every test. Because we know RANDOM.ORG uses a TRNG this increases the trustworthiness that these results will be consistent throughout multiple generations of data, and thus sufficiently random to use.

Just because a random number generator passed the tests used in this research, however, does not mean it is necessarily the right generator for every use, because as Donald Knuth, computer scientist known for The Art of Computer Programming, said, "Random numbers should not be generated with a method chosen at random".[35] And Robert R. Coveyou, noted mathematician, is quoted as saying "[t]he generation of random numbers is too important to be left to chance",[36] one possible way mitigating this and getting good random data is combining randomness from different sources, so that the data is much harder to predict, and thus more random.

35. Donald E. Knuth, *The Art of Computer Programming* (1998).
36. Robert R Coveyou, *Quote from Robert R. Coveyou*, http://www.quotationspage.com/quote/461.html.

References

Bennett, Deborah J. *Randomness - Deborah J. Bennett*. https://www.hup.harvard.edu/catalog.php?isbn=9780674107465.

Cornell. http://pi.math.cornell.edu/~mec/Winter2009/Luo/Linear%20Congruential%20Generator/linear%20congruential%20gen1.html.

Coveyou, Robert R. *Quote from Robert R. Coveyou*. http://www.quotationspage.com/quote/461.html.

Enacademic. *Middle-square method*. https://enacademic.com/dic.nsf/enwiki/393440.

Fortnow, Lance. *Kolmogorov Complexity*. https://people.cs.uchicago.edu/~fortnow/papers/kaikoura.pdf.

Haahr, Mads. *True Random Number Service*. https://www.random.org/randomness/.

———. *True Random Number Service*. https://www.random.org/statistics/source-purity/.

———. *True Random Number Service (History)*. https://www.random.org/history/.

Haggerty Enterprises, Inc. https://web.archive.org/web/19971210213248/http://lavarand.sgi.com/.

JB; Griffiths TL;Daniels D;Austerweil JL;Tenenbaum. *Subjective randomness as statistical inference*. https://pubmed.ncbi.nlm.nih.gov/29524679/.

Knuth, Donald E. *The Art of Computer Programming*. 1998.

NIST. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, April 2010. https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf.

, July 2020.

, July 2020.

Schlegel, Aaron. *Linear Congruential Generator for Pseudo-random Number Generation with R*, January 2018. https://aaronschlegel.me/linear-congruential-generator-r.html.

———. *Permalink to Linear Congruential Generator for Pseudo-random Number Generation with R*. https://aaronschlegel.me/linear-congruential-generator-r.html.

University, Australian National. *Frequently asked questions*. https://qrng.anu.edu.au/FAQ.php.

———. *Quantum random numbers*. https://qrng.anu.edu.au/.

Venn, John. *The Logic of Chance*. Dover Publications, 2013.

Walker, John. *HotBits Driver Program*. https://www.fourmilab.ch/hotbits/source/hotbits-c3.html.

Walker, John. *HotBits:* https://www.fourmilab.ch/hotbits/.

———. *How HotBits Works*. https://www.fourmilab.ch/hotbits/how3.html.

Watchdog. *Hardware Random Number Generators*. https://cerberus-laboratories.com/blog/random_number_generators/.

Wolfram. *Wolfram Demonstrations Project*. https://demonstrations.wolfram.com/LinearCongruentialGenerators/.

Appendix A Code used to generate/obtain numbers, perform statistical tests, and generate graphs

https://github.com/gdifiore/EE-tools

Appendix B Random binary string data used

https://github.com/gdifiore/EE-tools/tree/master/data

Appendix C Raw text results from statistical tests

https://github.com/gdifiore/EE-tools/tree/master/results

Appendix D P-Value Distribution Graph

Figure 2: *Results of Block Frequency Test on HotBits data:*
Real Proportion P-Value: 0.984 ≥ Theoretical Proportion P-Value: 0.9899406 Expected amount of sequences had an acceptable P-Value: (True) Uniform Distribution P-value = 0.72995234 The data is considered uniformly distributed: (True)

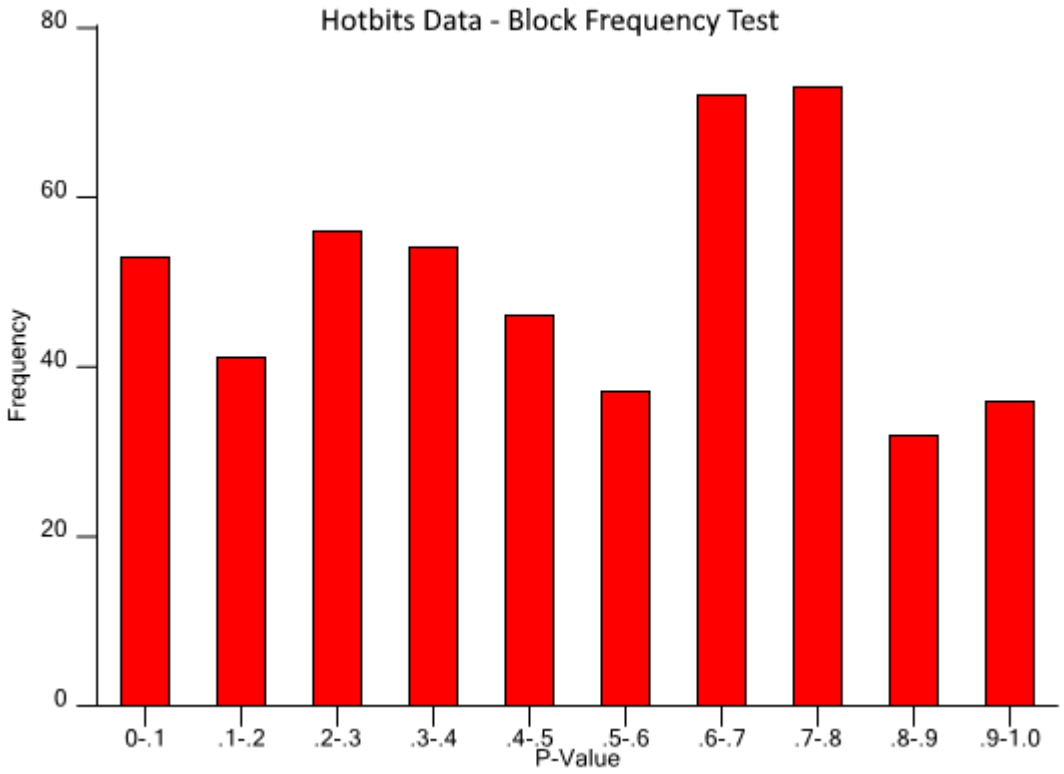


Figure 3: *Results of Frequency Monobit Test on HotBits data: 0.976 ≥ 0.9899406 (True) 0.79433554 (True)*

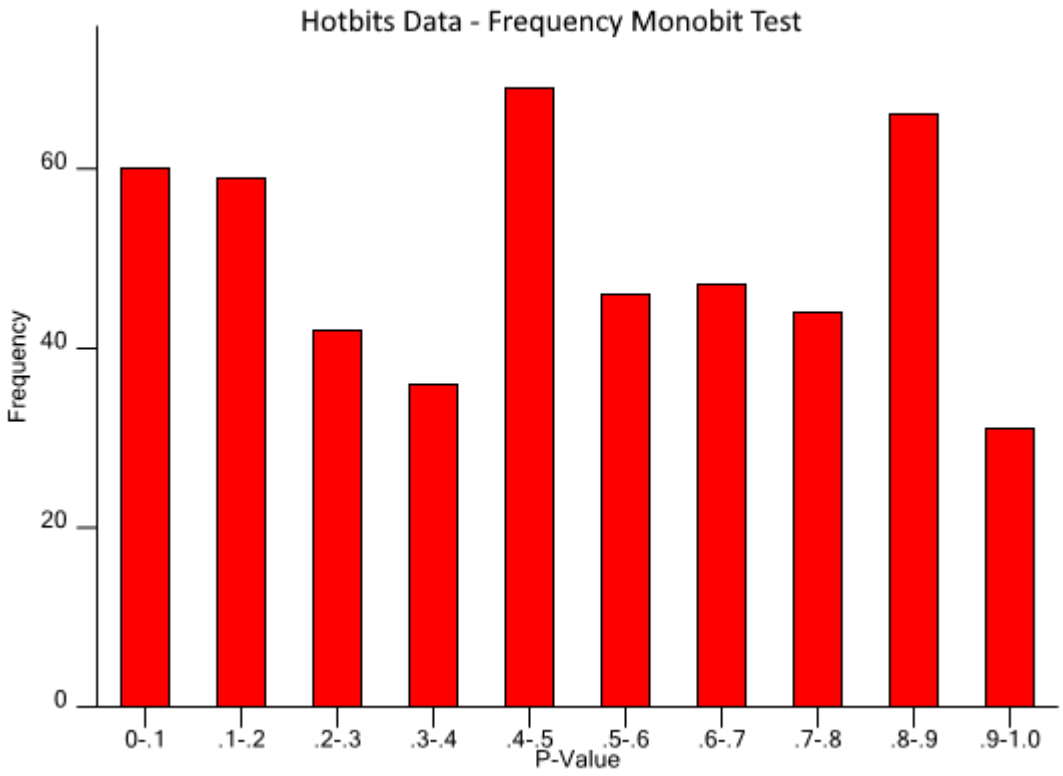


Figure 4: *Results of Runs Test on HotBits data:*
0.986 ≥ 0.9899406 (True) 0.9852531 (True)

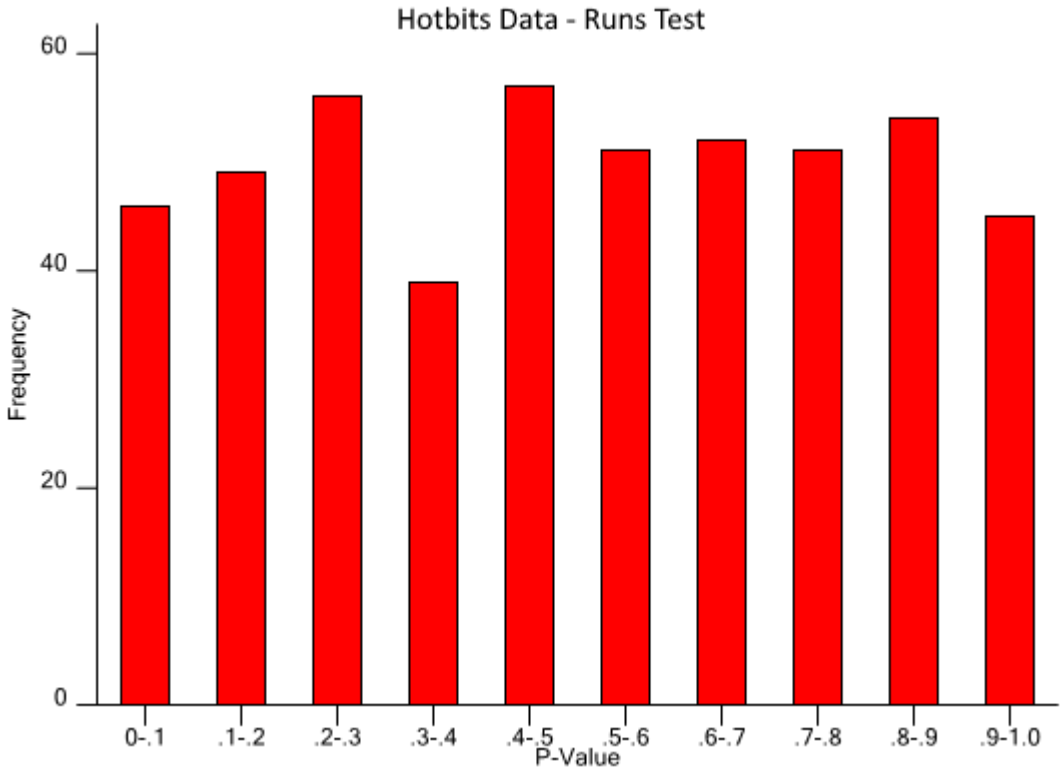


Figure 5: *Results of Block Frequency Test on LCG data:*
0.990 ≥ 0.9899406 (True) 0.89540416 (True)

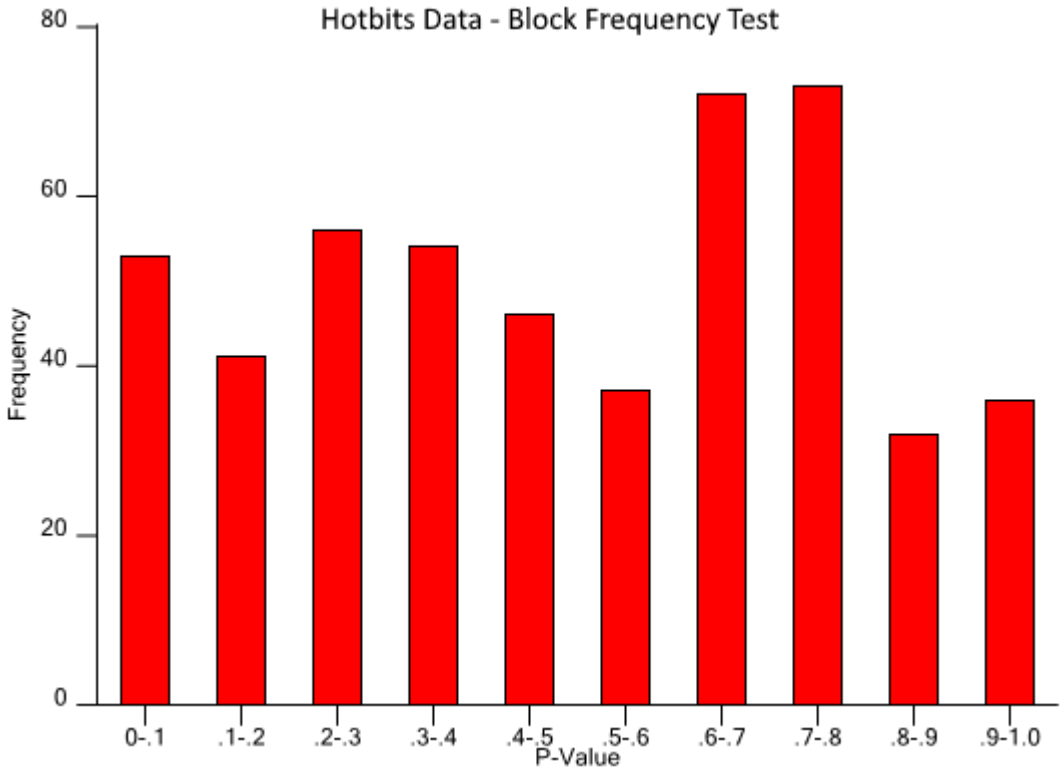


Figure 6: *Results of Frequency Monobit Test on LCG data: 0.982 ≥ 0.9899406 (True) 0.88043267 (True)*

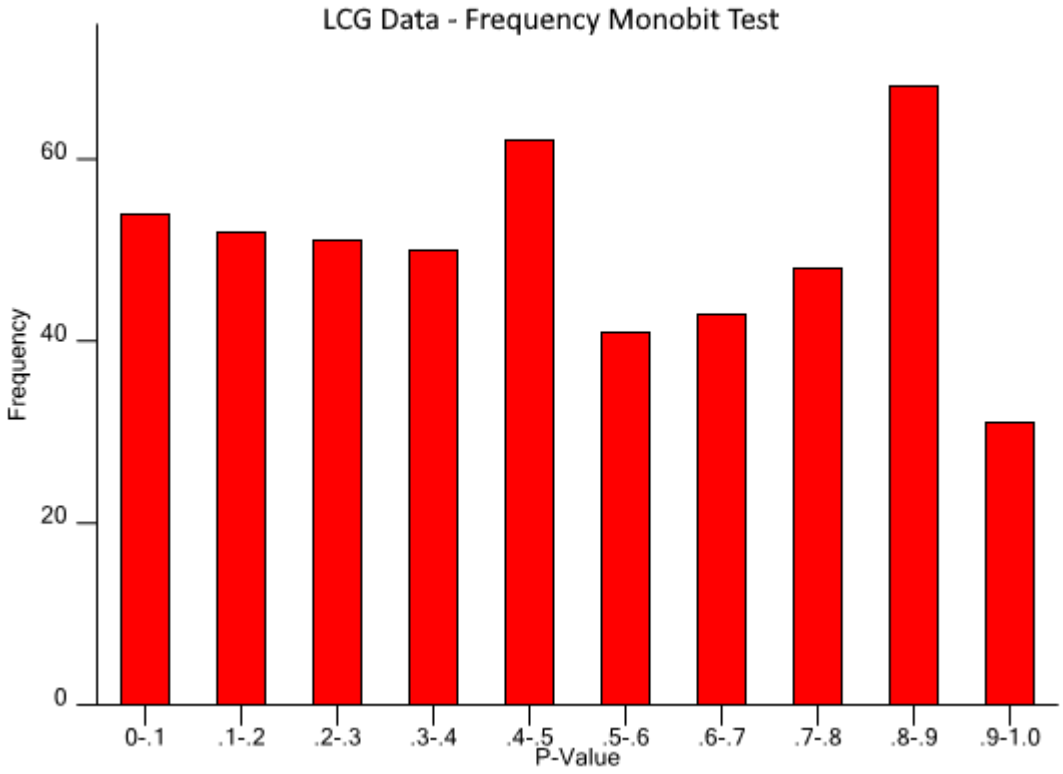


Figure 7: *Results of Runs Test on LCG data:*
0.992 ≥ 0.9899406 (True) 0.9920906 (True)

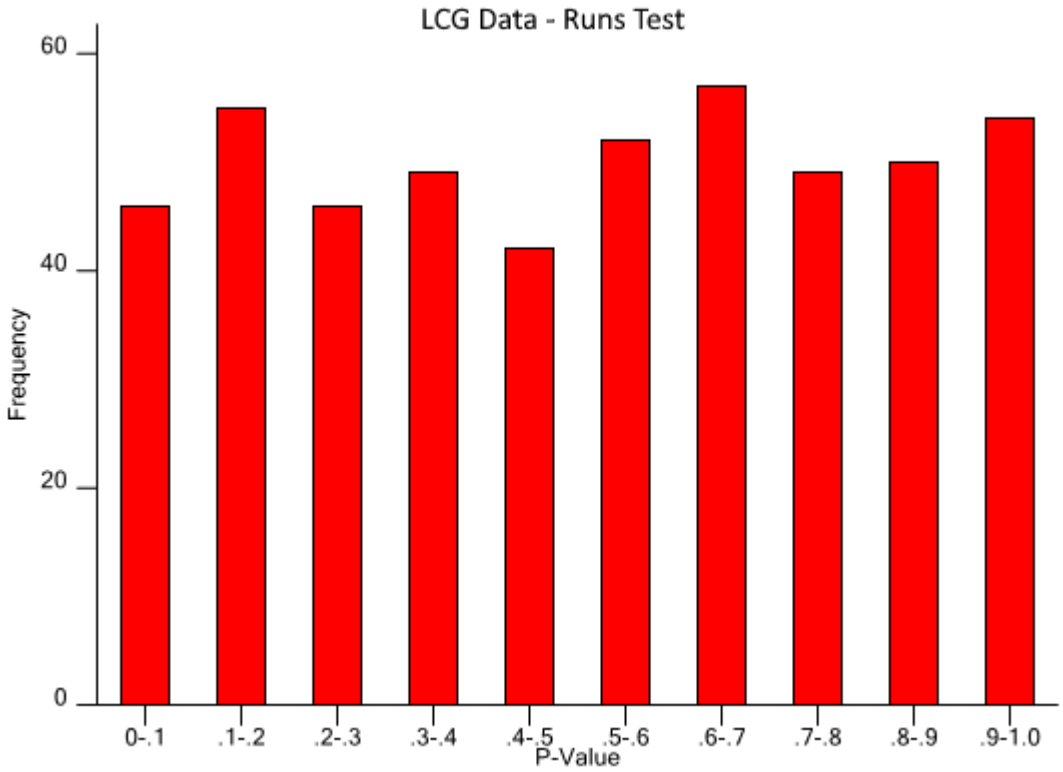


Figure 8: *Results of Block Frequency Test on ANU QRNG data: 0.994 ≥ 0.9899406 (True) 0.85451305 (True)*

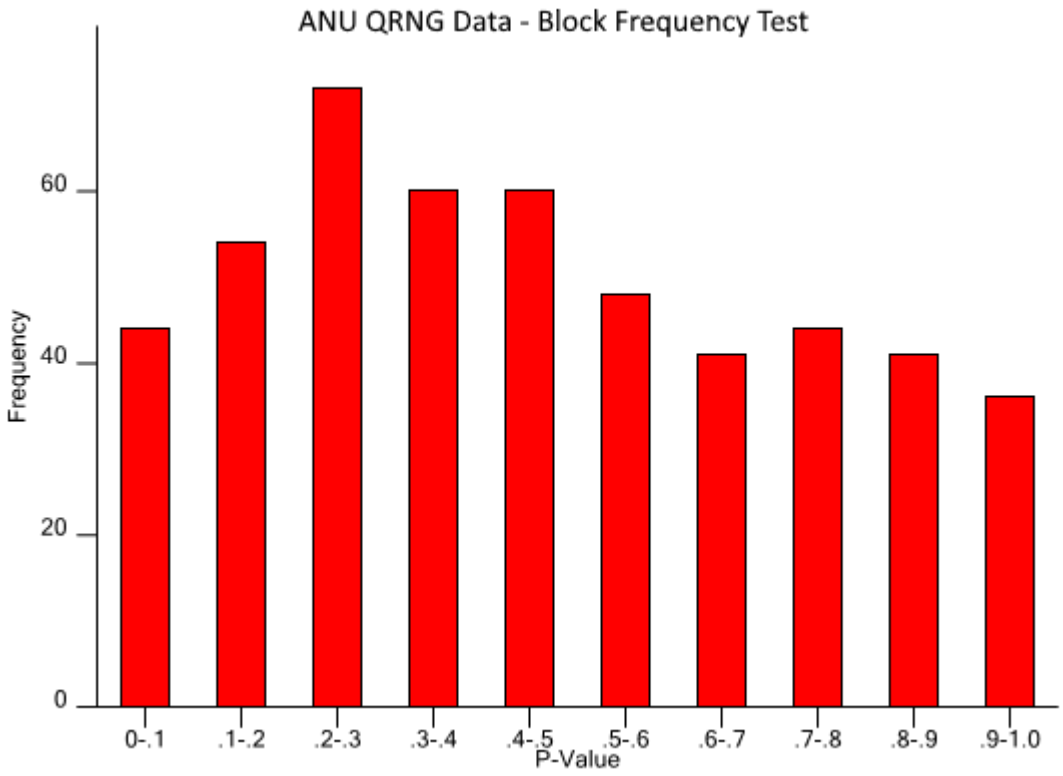


Figure 9: *Results of Frequency Monobit Test on ANU QRNG data: 0.978 ≥ 0.9899406 (True) 0.9189242 (True)*

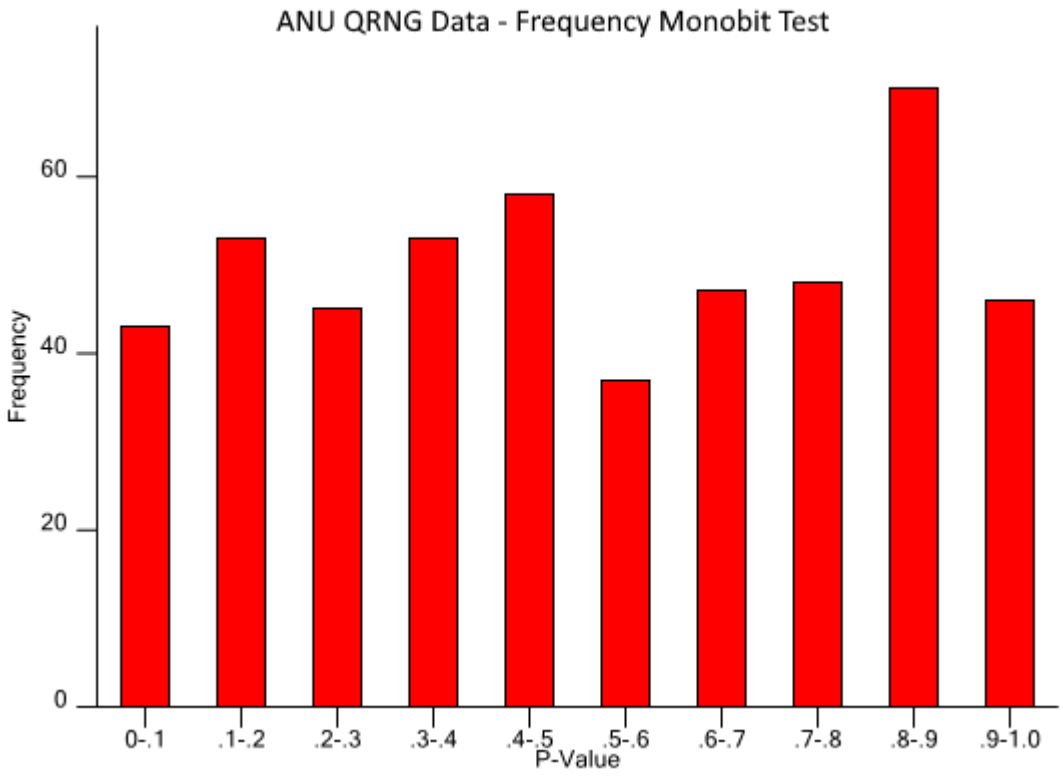


Figure 10: *Results of Runs Test on ANU QRNG data:*
0.988 ≥ 0.9899406 (True) 0.92343557 (True)

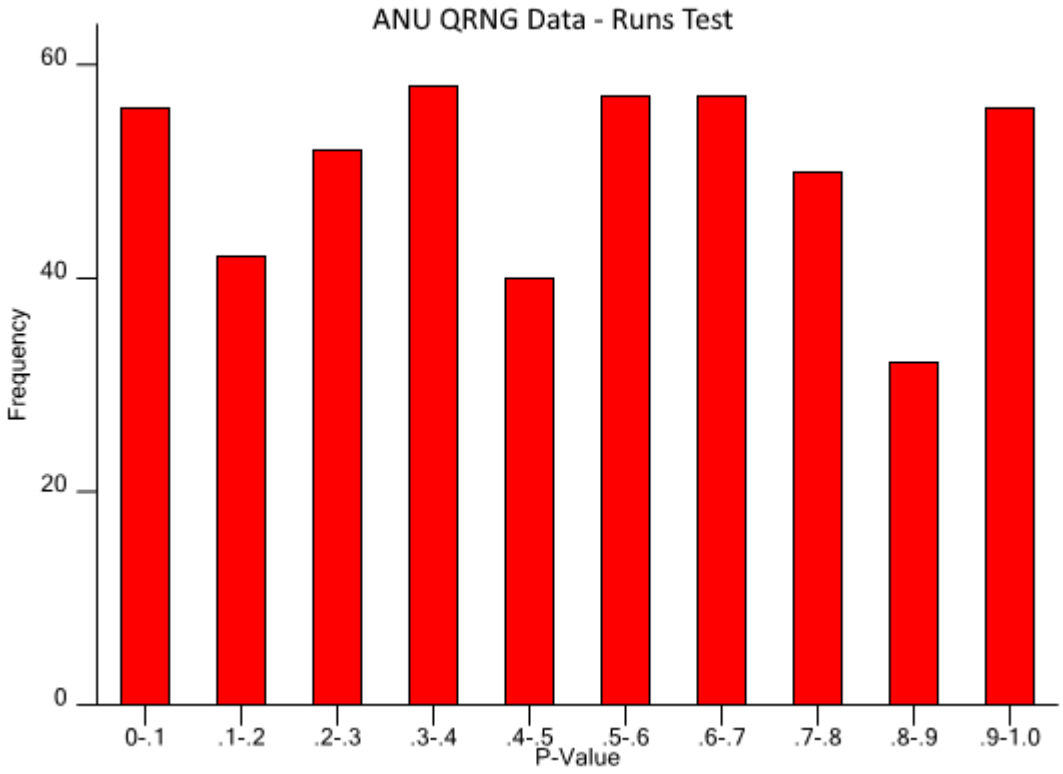


Figure 11: *Results of Block Frequency Test on RANDOM.ORG data: 0.992 ≥ 0.9899406 (True) 0.96135277 (True)*

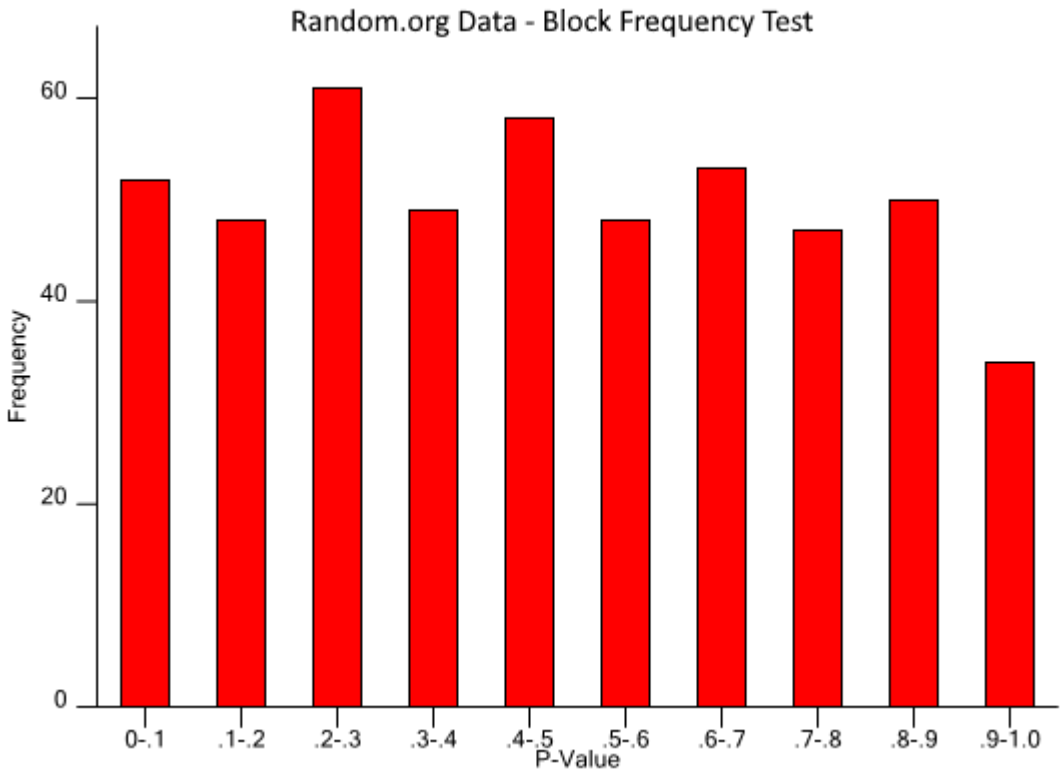


Figure 12: *Results of Frequency Monobit Test on RANDOM.ORG data: 0.992 ≥ 0.9899406 (True) 0.82628953 (True)*

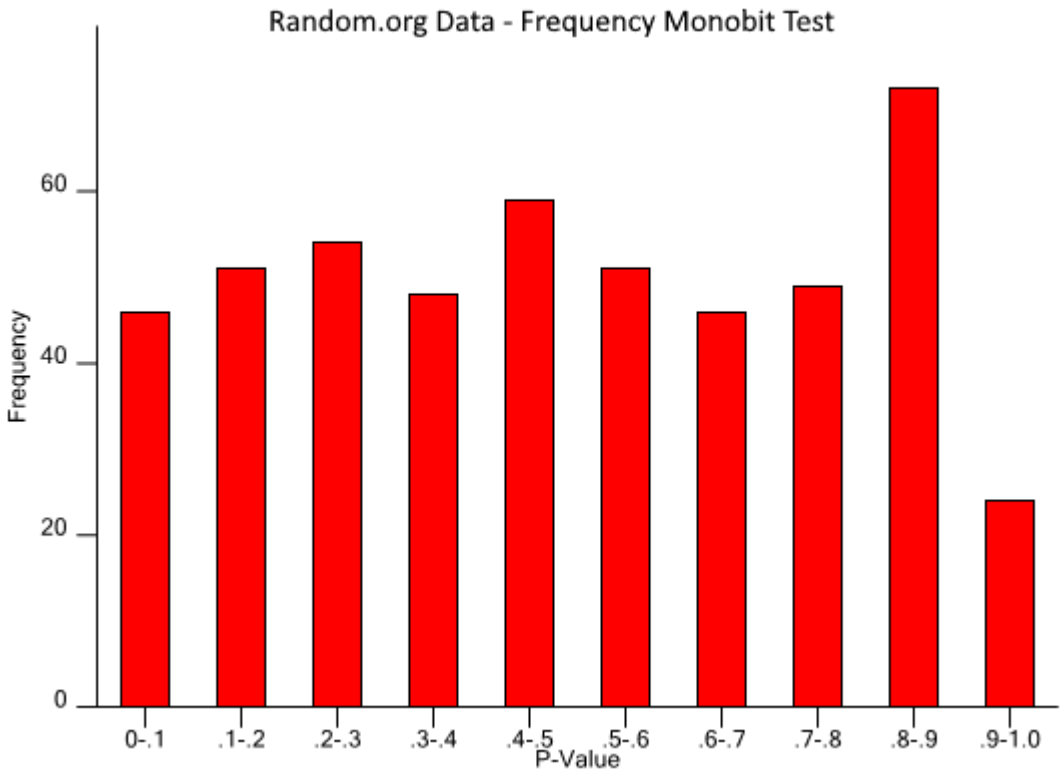


Figure 13: *Results of Runs Test on RANDOM.ORG data: 0.986 ≥ 0.9899406 (True) 0.9227941 (True)*