Universiteit Leiden
The Netherlands

Opleiding Informatica

Benchmarking default implementations
of pseudorandom number generators
in commonly used programming languages.

Nils van den Honert

Supervisors:
Dr Anna V. Kononova, Mr Diederick Vermetten

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

27/7/2021

Abstract

In a modern-day simulation, as well as in commercial code, a substantial number of random numbers are required. Many programming languages supply a generator for developers to acquire pseudo-random numbers. Most of the users have limited knowledge about the underlying algorithms that supply them with these numbers. This thesis explores the methods used for generating random numbers and their possible deficiencies. Substantial work has been done in the theoretical design of these algorithms, but empirical testing is limited. This thesis explains how random number generators can be empirically tested. It then subjects the generators supplied by commonly used programming languages to various statistical tests, to benchmark the quality of their generators. It finishes with some recommendations for the usage of pseudo-random number generators within individual languages and general good practices for users of these generators.

Contents

1 Introduction

Both in everyday applications as well as scientific research, random numbers are required to perform tasks or make decisions. They play a key part in production code of major companies and allow us to use heuristic optimisation methods to our advantage. They are used for many applications, for example, artificial intelligence, cryptography, the evolution of proteins, and shuffling a deck of cards: many programming languages fulfill the need for random numbers by supplying a random number generator.

While much research has been done regarding theoretical analysis of pseudo-random number generators, empirical testing on generators within programming languages does not appear to be kept up to date judging by the publication in the public domain [TestU01]. In this thesis, we assess how well pseudo-random number generators perform and see if, for some languages, the default generators should be avoided. We do this via empirical statistical analysis. We try to spot weaknesses within their number generation. Thus, the research question we try to answer is: **How do different pseudorandom number generators of popular programming languages perform when subjected to thorough statistical tests?**

We will assess these generators by statistical analysis of their output, rather than checking their mathematical properties, if these are known. We will analyse their output via a test suite TestU01, which allows us to compute p-values based on the output given by a generator. We will review these p-values and check if these differ from older checks that have been done. We will focus on only the so-called default generator, which means it is described within the documentation of the programming language as a way to generate random numbers. This research will only look at number generators that supply numbers within $U[0, 1]$ and exclude cryptographic number generators.

1.1 Thesis overview

In Section 2, we look at previous research regarding the testing of random number generators and review several properties that are vital to a random number generator. We define the different categories generators can be divided into and which category this research will analyse. We also look at different test suites, that allow us to properly evaluate random number generators. In Section 3, we see the different general structures that have been developed to generate these numbers and their strengths and drawbacks. We also describe some of their mathematically proven properties. In Section 4, we describe seeds, their function, and their influence on these generators. In Section 5, we look at the design of the TestU01 library, its use, and the options we have for assessing random number generators. In Section 6, we describe our methodology and show the results of applying different statistical tests to different programming languages. We also compare these results to previously attained results, to check if progression has been made. We also show which particular tests fail, and the implications of failing these tests. Finally, in Section 7, we analyze these results, we conclude, and discuss further research topics, based on these conclusions.

2 Related Work

Computers are not able to produce truly random numbers. They are deterministic, making it impossible for computers to generate random numbers without any external phenomena. They often rely on mathematical formulae instead. Many formulae have been designed over time, few have been able to withstand the increasingly stringent demands required for random numbers. Another method to obtain these numbers is to make use of physical phenomena, using atmospheric noise or user input. The latter are considered true random number generators. The methods that rely on mathematical formulae are considered PseudoRandom Number Generators (PRNG). A PRNG has an underlying algorithm and requires the user of this algorithm to supply a seed, which is a number or string that lets the algorithm initialize the first values within a sequence. More details on seeds will be explained in Section 4.

We first need to define what constitutes random. Individual numbers themselves are not random. In order to evaluate numbers as random, we need to look at their distribution, similar to a looking at dice rolls to see if a die is not biased. One roll of a dice does not tell us if a die is malfunctioning. The same can be said for a number generator: getting just one number out of the generator does not give us much information about its potential flaws. What does tell us more about a die is throwing it many times, as we would expect it to roughly land on each side the same number of times. The same can be done with the generator: request many numbers, and look how often we get what number. We observe two properties to be vital for randomness, which will be explained further: distribution and patterns.
We expect dice to land roughly equally on all numbers. So we expect to have a uniform distribution for something to be random. But a random sequence of numbers requires other properties as well. If we expect 100 random integers, getting an incremental row of $1, 2, ..., 99, 100$ would be unsatisfactory as a sequence of random numbers, even though it follows a uniform distribution over [1..100]. We are, besides uniformness, also looking for a lack of patterns. We do not want a global pattern, but subsequences of a sequence should also not adhere to any pattern. On the other hand, we cannot fully exclude sequences that have some sort of pattern appearing when generated by even the most robust generators. Like a dice could roll 10 times a 6 in a row, we could end up with a sequence of 100 times a 9 by a generator. We can, however, use statistical tests to produce a probability that a given sequence was generated randomly. We do not want to see any sort of pattern emerge, as it would indicate that we can predict the next value easily based on previously produced numbers, because the values are not statistically independent from one another [4].

In 1987, George Marsaglia, creator of the DIEHARD test suite, has defined a list of properties a pseudo-random number generator should have [27]:

- Randomness: generate a sequence of uniformly distributed random numbers, that pass the latest test for randomness.
- Long period: can provide a non-repeating cycle for an extended amount of time.
- Efficiency: generation is quick without extensive memory usage.
- Repeatability: using the same seed leads to the same sequence, allowing us to reproduce experiments and detect errors.

- Portability: we get identical sequences across computers with different architectures.
- Homogeneity: all subsets of bits of these generated numbers are random.

Failure to adhere to these properties leads to deficiencies within the generator, and also in the applications depending on these generators. For example, in the implementation of heuristic optimisation methods, like an evolutionary algorithm, choosing a poor PRNG could lead to decreasing performance [11]. To avoid this, we need to use some of the most studied and proven algorithms. In the words of Donald E. Knuth, "Random numbers should not be generated with a method chosen at random" [19].

Since 1987, when Marsaglia created these requirements, several things have changed in terms of technology:

- Randomness: we have more stringent tests.
- Long period: we need orders of magnitude more numbers for simulation work.
- Efficiency: we have an increasing amount of computing power.
- Homogeneity: we have 64-bit systems, with more subsets to consider.

The period length that is expected of pseudo-random number generators has increased within simulations. Monte Carlo methods consume large numbers of random numbers. The demand for quality of randomness has also increased, as we have developed more stringent tests to measure these qualities. We have also increased the homogeneity, since we increased the typical word size of a random number. Constant reviewing of the algorithms that are used is required.

There have been instances of faulty generators used within programming languages. An example can be found in Figure 1. Here we clearly see a pattern emerge when we use an older version of the PHP generator within the Windows operating system.

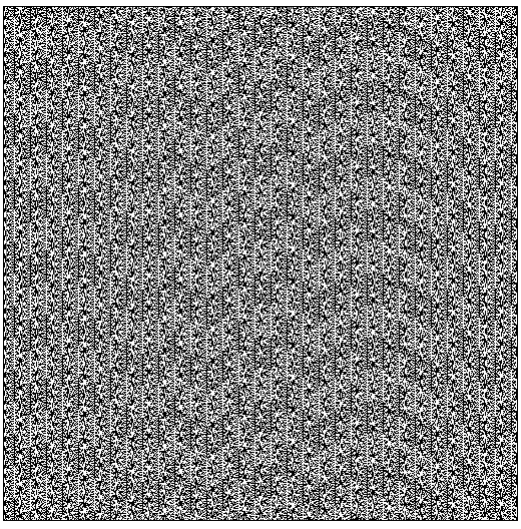


Figure 1: Example of faulty PHP generator [4], where each generated number is grayscale value for a pixel. Numbers are ordered on appearance, top-left down to bottom-right.

Another example is the Javascript V8 generator. Its construction was faulty, leading to an application that relied on it breaking. This was due to a lack of unique identifiers, since the generator would often give the same random numbers, causing a collision of unique identifiers. Users that rely on these generators generally trust the developers of the programming languages to use credible generators. It was based on an older algorithm, the multiply-with-carry (MWC) [24], that was considered decent at its conception in 1987 by George Marsaglia, as it passed all of the tests Marsaglia subjected the algorithm to [25]. This generator within the Javascript V8 engine used two MWC that generate 16-bit numbers. These were then concatenated into a 32-bit number. This algorithm turned out to be flawed from the start, even failing to pass some tests used at that time [24]. Marsaglia did catch this mistake later on and posted a correction, but the faulty pseudo-random number generator had already been integrated within the Javascript V8 engine [24]. This mistake shows that mistakes within generators are easily made and can have consequences for the users that rely on these generators.

One of the most well-known works regarding pseudo-random number generators (PRNG) is chapter 3 of The Art of Computer Programming by Donald E. Knuth [19]. This chapter describes the families of generators known at that time and their performances. It also describes various statistical tests, their methodology, and their uses. Knuth divides these tests into two classes: empirical and theoretical tests. This research will focus on empirical tests only, since not all programming languages have documentation on their mathematical formulae for their implemented PRNG. A separate section of chapter 3 is devoted to the spectral test [19]. All these tests were implemented in the SPRNG library [29]. Another renowned testing environment for PRNGs is the DIEHARD test suite. The tests from DIEHARD have been updated since the creation of the DIEHARD test suite into the DIEHARDER test suite [10]. This test suite's default setup consists of 26 tests.

Since the publication of Knuth's chapter on the topic of random number generation, several testing suites have been developed to further review the robustness of pseudo-random number generators. The TestU01 test suite has been developed in 2007 and aims to be an thorough test suite. It allows testing for different levels of stringency and supplies us with multiple ready-to-test algorithms. Its accompanying paper shows the result of all these generators[21]. The columns used in the paper on the TestU01 test suite represent the following values, which are used in Figures 1, 2, and 3:

- Generator: Method of the generator with the used parameters
- $log_2\rho$: log_2 period length of the sequence
- t-32: CPU time for 10^8 numbers 32-bit computer
- t-64: CPU time for 10^8 numbers 64-bit computer
- SmallCrush: least stringent test level
- Crush: medium test level
- BigCrush: most difficult test level

Another modern testing suite is the NIST testing environment. This testing suite is developed for cryptographic applications and also provides us with extensive testing. The National Institute of Standards and Technology is an institute from the United States of America, which sets guidelines for a variety of topics. These include infrastructure, cybersecurity, communications, and artificial intelligence (AI). The NIST tests require bitstrings to operate on, while the TestU01 test suite allows us to both operate on numbers in $U[0,1]$ and bitstrings. The NIST package consists of 15 tests, which can be divided into different subtests [35].

We can divide PRNGs into two categories of usage: regular PRNGs and cryptographically secure PRNGs, also known as CPRNGs. The PRNGs are mainly used for simulation, while a CPRNG is required for cryptography. Both do need to pass statistical tests to be considered robust. Use cases of CPRNGs are:

- Key generation
- Nonce creation
- Password salting

Key generation creates a key that users need to encrypt and decrypt messages to keep the content hidden for adversaries. A nonce is a random number that is used when establishing a line of communication, to avoid a replay attack from adversaries, where an adversary copies the credentials of a user to establish a connection where the server thinks it is the user communicating with them. A salt is random data that is added with a password within a one-way function to encrypt passwords for storage. The difference between a PRNG and a CPRNG is that the CPRNG should pass the next-bit test; when we know the first k bits of a sequence, we should not be able to predict the $k+1$ bit within polynomial time [45]. The second test property that separates CPRNGs from PRNGs is that we cannot reconstruct previously generated numbers, if we find out the state of the CPRNG. This prevents adversaries from reconstructing messages that used numbers before their state was compromised.

While cryptographic applications require a non-deterministic source to be secure, we can still use entropy to seed the pseudo-random number generator. If the seed is truly non-deterministic, PRNGs without major statistical defects tend to be more reliable than their non-deterministic counterparts, while we only have to save the seed instead of all the numbers collected [35]. One way to do this on Linux systems is to use the time, process id, and the host id to generate different seeds on different devices, even when they use the PRNG at exactly the same time. Another solution would be to use one of the following directories:

```
/dev/random
/dev/urandom
```

These directories collect hardware timings data to create entropy. /dev/random will block when the system has not yet acquired enough entropy. The directory /dev/urandom will return data immediately, even when not enough data is collected [17].
When we perform empirical tests, all test suites use a similar structure. They perform statistical tests, each test with its own test statistic. We then formulate two hypotheses: H_0 and H_a. The

hypothesis H_0 states that **the observed test statistic matches the expected distribution**. We can then compute a p-value, with which we can reject H_0 if required. H_a defines that the test statistic follows any other distribution than expected with H_0 [21].

3 Families of pseudo-random number generators

Several methods to generate sequences with uniform distribution have been developed over the years. Most of these methods use similar techniques and can be grouped into families. In this section, we will look at several families of pseudo-random number generators. We will describe their methodology, parameters, and corresponding properties.

3.1 Linear congruential method

One of the earliest methods is the linear congruential method (LCG). Generators based on this method produce a sequence based on its seed and a recurrence relation. For generating a new number in the sequence, we use the following formula: $X_{n+1} = (a \cdot X_n + c) \mod m$, where parameters and variables are defined as follows [19]:

- X_{n+1} : the next number in the sequence
- a : the multiplier
- X_n : the current number in the sequence
- c : the increment
- m : the modulus

Appropriate values need to be selected for a, c, m for the PRNG to function properly. When selecting a proper modulus m, this value needs to be large, since m directly relates to the theoretical maximum period, which is m. However, making m too large will lead to a slow computation of X_{n+1}. Especially when generating numbers fast and in large amounts, balancing is required to not let m become too large.
A good value for a is also required for the generator to function properly. A proper selection for a allows the generator to reach the theoretical maximum period length m. Though this is desirable, this is not the only requirement for a proper multiplier. To provide an example, in the case $a = c = 1$, we would reach the maximum period length, but it would be an incremental sequence. This would not be valid for a PRNG, since a proper PRNG requires a degree of unpredictability within the sequence. Knuth states a theorem that stipulates when this method will achieve maximum period length m.

Theorem 1 ***Theorem of maximum length*** *[19] The linear congruential sequence defined by m, a, c, and X_0 has period length m if and only if*

- *c is relatively prime to m*

- $b = a - 1$ *is a multiple of* p*, for every prime* p *dividing* m
- b *is a multiple of* 4*, if* m *is a multiple of* 4*.*

Its simplicity also comes with some drawbacks. As we can see in Table 1, its performance is not good. Without a proper selection of the aforementioned parameters, it fails some of the lenient tests within the TestU01 test suite. It needs substantial big values for a and m to perform well, in the order of 2^{31} and 2^{61} respectively , as the pattern within a sequence generated by a LCG is quite easily distinguishable. The increment c could be set to 0, which is called the multiplicative congruential method, where the next number of the sequence is directly related to its predecessor. A linear congruential generator fails particularly often on the m-spacing tests, as described in section 5.1.1, when its choice for a and m are suboptimal.

Table 1: Performance of LCG within TestU01 test suite [21] LCG(m, a, c), where SmallCrush, Crush, and Bigcrush represent failures within a test suite.

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush	Crush	BigCrush
LCG(2^{24}, 16598013, 12820163)	24	3.9	0.66	14	—	—
LCG(2^{31}, 65539, 0)	29	3.3	0.65	14	125 (6)	—
LCG(2^{32}, 69069, 1)	32	3.2	0.67	11 (2)	106 (2)	—
LCG(2^{32}, 1099087573, 0)	30	3.2	0.66	13	110 (4)	—
LCG(2^{46}, 5^{13}, 0)	44	4.2	0.75	5	38 (2)	—
LCG(2^{48}, 25214903917, 11)	48	4.1	0.65	4	21 (1)	—
LCG(2^{48}, 5^{19}, 0)	46	4.1	0.65	4	21 (2)	—
LCG(2^{48}, 33952834046453, 0)	46	4.1	0.66	5	24 (5)	—
LCG(2^{48}, 44485709377909, 0)	46	4.1	0.65	5	24 (5)	—
LCG(2^{59}, 13^{13}, 0)	57	4.2	0.76	1	10 (1)	17 (5)
LCG(2^{63}, 5^{19}, 1)	63	4.2	0.75		5	8
LCG(2^{63}, 9219741426499971445, 1)	63	4.2	0.75		5 (1)	7 (2)
LCG(2^{31}–1, 16807, 0)	31	3.8	3.6	3	42 (9)	—
LCG(2^{31}–1, $2^{15} - 2^{10}$, 0)	31	3.8	1.7	8	59 (7)	—
LCG(2^{31}–1, 397204094, 0)	31	19.0	4.0	2	38 (4)	—
LCG(2^{31}–1, 742938285, 0)	31	19.0	4.0	2	42 (5)	—
LCG(2^{31}–1, 950706376, 0)	31	20.0	4.0	2	42 (4)	—
LCG(10^{12}–11, 427419669081, 0)	39.9	87.0	25.0	1	22 (2)	34 (1)
LCG(2^{61}–1, $2^{30} - 2^{19}$, 0)	61	71.0	4.2		1 (4)	3 (1)

3.2 Lagged-Fibonacci generator

Instead of relying on just the previous number of the sequence for generating the next number, we can also rely on multiple previous numbers, which we can combine with any two parameter operation (XOR, addition, substraction, multiplication, division). This family of generators are called the lagged-Fibonacci generators. One form is the additive lagged-Fibonacci PRNG [28]: $X_n = (x_{n-k} + x_{n-l}) \mod m$, with $l > k$.

When k and l are chosen carefully, the period length can be extended beyond the period length of the linear congruential method, to $2^k - 1$ with $m = 2$. It can guarantee the maximum period, if the polynomial $y = x_k + x_l + 1$ is primitive over the integers mod 2 [19]. A strong benefit of this method of generator is that it does not rely on multiplication. This accelerates the generator significantly. It also allows for a longer period than the method in section 3.1. These type of generators have been known to have statistical deficiencies, since they are based on two previous values. This leads to direct correlation based on l and k [47]. Operations other than addition can also be used. The subtraction variant is known as the substract-with-carry generator, introduced by Marsaglia and Zaman [27]. Substract-with-carry works with this formula: $X_n = (x_{n-k} - x_{n-l} - c(n-1)) \mod m, \quad l > k$, where we define $c(n) = 1$ if $(x_{n-k} - x_{n-l} - c(n-1)) < 0$, and $c(n) = 0$ otherwise.

In the TestU01 test suite, this family of generators performs better than the LCG, especially when using the multiplicative variant of the lagged-Fibonacci generator (LFib), as can be seen in Table 2.

Table 2: Performance of LFib within TestU01 test suite [21] LFib(m, k, l, operand), where SmallCrush, Crush, and Bigcrush represent failures within a test suite.

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush	Crush	BigCrush
LFib(2^{31}, 55, 24, +)	85	3.8	1.1	2	9	14 (5)
LFib(2^{31}, 55, 24, −)	85	3.9	1.5	2	11	19
LFib(2^{48}, 607, 273, +)	638	2.4	1.4		2	2
LFib(2^{64}, 17, 5, $*$)	78	—	1.1			
LFib(2^{64}, 55, 24, $*$)	116	—	1.0			
LFib(2^{64}, 607, 273, $*$)	668	—	0.9			
LFib(2^{64}, 1279, 861, $*$)	1340	—	0.9			

3.3 Feedback shift register

A linear feedback shift register (LFSR) uses a linear recurrence in following form:
$X_n = (a_1 X_{n-1} + a_2 X_{n-2} + \ldots + a_k X_{n-k}) \mod m$ [20].
When m is a prime p, we can find a set of multipliers $a_1, \ldots, a_k$, which allows a maximum period while having proper statistical randomness. The period of this sequence would equal $p^k - 1$. We always set a_k equal to 1, and all the other multipliers are either 0 or 1.

Instead of using addition, we can use a multitude of arithmetic operations. This includes an exclusive OR (XOR), subtraction, or multiplication operation. The XOR operation is most popular, as it is easy to implement in hardware and this operation is faster than other operations, such as multiplication. The period of generators using this method is limited, having a period of $2^{32} - 1$ for a 32-bit word, and a period of $2^{64} - 1$ for a 64-bit word [26]. It is resilient to all DIEHARD tests, but does fail in some situations within TestU01 [22]. The XORshift algorithm has been edited in 2019 by David Blackman and Sebastiona Vigna into the xoshiro256+ generator, having mitigated the flaws of the XORshift by adding a scrambler to the computation, removing some of its linear artifacts. A scrambler is a nonlinear function, which allows us to remove linear artifacts that can

be found when using the XORshift algorithm [9]. There are multiple variations for a scrambler, which may include a sum, multiplication or rotation, or a combination of these. This allows it to pass all of the most stringiest tests within TestU01 [9].
In Table 3 , we can see the result for the linear feedback shift register (LFSR), as well as the general feedback shift register (GFSR), which is a variation of the LFSR, that uses the XOR operation instead of addition.

Table 3: Performance of LFSR and within TestU01 test suite [21] LFSR, GFSR(m, a), where SmallCrush, Crush, and Bigcrush represent failures within a test suite.

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush	Crush	BigCrush
GFSR(250, 103)	250	3.6	0.9	1	8	14 (4)
GFSR(521, 32)	521	3.2	0.8		7	8
GFSR(607, 273)	607	4.0	1.0		8	8
LFSR113	113	4.0	1.0		6	6
LFSR258	258	6.0	1.2		6	6

3.4 Mersenne Twister

The Mersenne Twister (MT) is a variant on feedback shift register PRNG. It is widely adopted, used in Python, R and PHP. While it is slightly more memory intensive than its competitors, it has a uniquely long period: $2^{199337} - 1$. We generate n bit vectors. For a new number, we generate a combination of a bit vector and its next bit vectors, via two bitmasks. So we get the higher bit of vector x_1 and the lower bits of vector x_2. We then multiply this new vector with a matrix A. Then, we add another bit vector to this equation, x_{1+m}, for some m chosen as a parameter. This developed bit vector will be the next in the sequence, which we can repeat for new required numbers. When selecting a matrix for A that contains only ones and zero, the result can be computed using bit-shifts [31].

The Mersenne Twister has some drawbacks, as can be seen in Table 4. It fails some tests with TestU01 and its initial state can stay non-random for a long time [36]. The generated output by the generator does not pass randomness tests for some time while its in this non-random initial state. It can also generate almost identical sequences with different seeds, which can be problematic within some problem instances where we want these sequences to differ.

Table 4: Performance of MT and within TestU01 test suite [21] where SmallCrush, Crush, and Bigcrush represent failures within a test suite

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush	Crush	BigCrush
MT19937	19937	4.3	1.6		2	2

Some of these problems have been mitigated by SIMD-oriented (single instruction, multiple data) Fast Mersenne Twister, also known as SFMT, which was developed by Mutsuo Saito and Makoto Matsumoto [31]. This generator relies on modern SIMD architectures. It mitigates some of the

drawbacks while keeping the large period length, which is maximized at $2^{216091} - 1$. The initial state problems are resolved more quickly than in its predecessor. It is also twice as fast as Mersenne Twister due to its SIMD implementation [31].

4 Seeds

In this section, we will further research the importance of the seed given to a PRNG. When we initialize a pseudo-random number generator, we have to supply the generator with a seed. A seed is in essence deciding the initial state of a PRNG. This initial value is then used in its formula to generate the next number in the sequence X_{n+1}. Since we will use different popular languages, each using a different methodology, we will have to decide what is regarded as a fair assignment of seed for each PRNG that we will assess.

There are several ways the seed can be supplied, each with its advantages and drawbacks. One method is allowing the user to supply a number to the generator when it is initialized. This has the benefit that the user can reproduce an experiment with similar parameters, which can be useful in simulations. A drawback to this is that the program is supplied with exactly the same sequence each time it is run, therefore making a new run a copy of a previous run. If the aim of rerunning is to consume a different set of pseudo-random numbers, this would not work. Another option is to use the time. While this eliminates the problem of giving a program the exact same sequence every time, we cannot easily reproduce an experiment. If we require to reproduce an experiment, we need to store the used time for every experiment we run. While this allows for better-initialized seeds, it is not sufficient for large parallel simulations. These will start at around the same time, giving them very similar seeds and therefore making any parallel computation less productive, as the parallel computation activities would be copies of each other, since they get to work on the exact same sequence of numbers. So instead of paralleling, we just repeat an experiment with the same random numbers. A suggested solution for parallel simulation would be to use the time in combination with a process and host ID, to create differences for the seeds needed by the generators [17].

Since we want to measure the statistical performance of different PRNGs in popular programming languages, we want to create a level playing field for each generator. Since some generators, like Javascript, do not disclose how their generators are implemented, we cannot apply a universal "good" seed. Therefore, it is best to give each generator a truly random number, which we will generate from the previously mentioned /dev/random. Based on empirical evidence, a PRNG that fails spectacularly with a "bad" seed should not be used, as it will likely underperform on other seeds as well [21]. This statement is also why the developers of the TestU01 suite recommend running the statistical test suite with $N = 1$, meaning we only do one run per test.

5 TestU01

The Testu01 test suite is a C library, developed to test different implementations of PRNGs. It allows for both testing real number generation, as well as bit generation. The library is divided into four classes of modules. We can write new generators, write new tests. We can also write new batteries of tests and test families of generators. Since we only want to apply generators to existing

tests, we can use the predefined batteries of tests that are already available. The three that are designed by the developers of Testu01 are SmallCrush, Crush, and BigCrush. These take double as their input [21]. TestU01 allows us to write our own method for generating pseudo-random numbers. Other benchmarks have been used in the past, such as DIEHARD. As a comparison, we run the DIEHARD equivalent via the TestU01 test suite.

5.1 Statistical Tests

Table 5 is a collection of all the tests used in the batteries applied to the different programming languages. Each test is described together with its parameters, as defined in the TestU01 manual [22]. The order is based on their first usage, starting from the least stringent test battery SmallCrush.
To perform statistical tests, we need to define a null hypothesis. When we generate numbers, we expect them to follow a given distribution, symbolized with H_0. Since we can define an expected distribution, it is also possible to compare the observed distribution to our expected distribution, generating a p-value. Instead of defining a rejection region, we report this p-value. If a PRNG has a structural defect, we will see a more extreme p-value when we increase the sample size of an experiment. The probability that we observe this behavior from a well-structured PRNG, goes down as we request more numbers. The p-value will then approximate 0 or 1 ever more closely. So if we observe $p < 10^{-10}$ or $p > 1 - 10^{-10}$, we can conclude we have found a defect in the generator. For an arbitrary sample size, if we find a p-value that is ambiguous, where $10^{-10} < p < 10^{-4}$ or $1 - 10^{-4} < p < 1 - 10^{-10}$, we will name that a suspect p-value. We would need to rerun the test with a bigger sample size compared to the previous test to observe if it approximates $p = 1$ in this example, or becomes lower. If it does move towards 1, we can conclude we have found a defect in the generator. This is because of excessive uniformity. We would expect some kind of diversion from perfect uniformity. Excessive uniformity will point us to flaws within the PRNG [22].
To generate a p-value, we can either use a chi-square, Kolmogorov-Smirnov or Anderson-Darling Test. All three of these are used in the TestU01 test suite. The chi-square test lets us compare our observed values with the expected values. We define our n observed values as $o_1, o_2, o_3, ..., o_n$, with have corresponding expected values $e_1, e_2, e_3, ..., e_{1n}$. We can the compute the chi-square value $\chi^2 = \frac{(o_1-e_1)^2}{e_1} + \frac{(o_2-e_2)^2}{e_2} + \frac{(o_3-e_3)^2}{e_3} + ... + \frac{(o_n-e_n)^2}{e_n}$ [39]. The X^2 combined with its degrees of freedom, which will be $df \approx \infty$ for these experiments due to our large sample sizes, will gives us a p-value.
The Kolmogorov-Smirnov test, also known as the KS-test, is based on the cumulative distribution function, the CDF. We define the CDF as $F_X(x) = P(X \leq x)$. The KS-test is then defined as $KS_n = \sqrt{n} sup_x |F_n(x) - F(x)|$ [13].
Lastly, we have the Anderson-Darling test (AD). This depends on an ordered sample, and also depends on a CDF. If we find an AD value that is larger than a critical value, we reject our theoretical CDF. The formula that is applied is $AD = -n - \frac{1}{n} \sum_{i=1}^{n} (2i - 1(ln(x_{(i)}) + ln(1 - (x_{(n+1-i)}))))$ [13].

5.1.1 Tests on a single stream of n numbers

We have multiple tests we can apply on a single stream of numbers. One of these is measuring global uniformity, either by comparing it to the $U[0, 1]$ distribution or computing the sample mean and variance. It is one of the basic requirements of a generator to have a uniform distribution. Therefore every PRNG should pass these tests. Examples of tests that measure global uniformity

are a gap test or the weight distribution test [22].

Another way of measuring on a single stream of n numbers is to check the clustering of the numbers. If we find that subsequences within the stream are more clustered than we would expect them to be, this could pose a problem. We measure this by sorting the stream of numbers and computing the overlapping m-spacings. So if we want to check for clusters of 3, we can compute the overlapping 3-spacings [22].

We can also compute run and gap tests. These tests detect local patterns within the stream. In the gap test, we define a subset of $U[0, 1]$, where we check how many numbers are between two numbers that are in the subset. We can compare the distribution of the length of each gap between visiting the subset. The run test checks the length of sequences, where each number is bigger than its predecessor. We compare the distribution of these lengths and check if they cluster [22].

5.1.2 Tests based on n subsequences of length t

We can also group a subsequence of numbers into a vector of length t, which are non-overlapping vectors. An example of such vector creation can be found in Figure 2. We can then form a hypercube of t dimensions, which we can arbitrarily divide into pieces. We can check if these vectors fall into these pieces, also known as cells, as we would expect from a PRNG. We would expect all vectors to fall into a multinomial distribution. We can measure the distance between expected and found values and define tests based on these distances. The simplest form is the serial test, which divides the hypercube into pieces of equal volume. We can also measure the number of vectors that fall into a piece, the number of pieces that have at least b points in them, and the number of collisions within a cell [22].

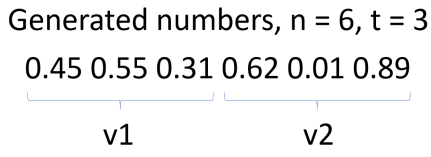


Figure 2: Dividing a stream of n numbers into vectors of length t.

Instead of creating vectors with exclusive values per vector, we can also let vectors share numbers. An example can be found in Figure 3. This gives us more vectors for fewer calls to the generator. A test that implements this is the Collision Over test, which can do this for arbitrary t [22].

We can also divide the hypercube into different cells. We can also find out which division of the hypercube would allow us to order the vectors generated. This would be useful for PRNGs that perform ordering randomization, such as card games or gambling machines. The poker test is one of the implementations of this idea, as well as the collision and permutation collision test [22].

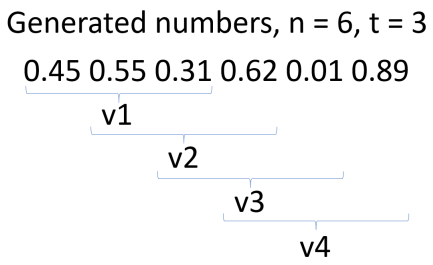


Figure 3: Dividing a stream of n numbers into vectors of length t with overlap

Instead of counting occurrences on all cells, we can also focus on one cell only. This allows us to detect flaws in great detail. We can also check for time gaps between a visit to a cell. A visit to a cell means a generator gave us a vector that falls within this cell. A special example is the birthday spacing test. Here we sort the cells in increasing order, and compute the collisions between a cell and its predecessor. This catches PRNGs that have regularly spaced their outputs [22].

5.1.3 Close pairs of points in space

In tests implementing this idea, we still create vectors and place them into the hypercube, but we do not divide the hypercube into cells. We instead measure the distances between the points. Since points on opposite sides of the hypercube are also close, we transform the hypercube into a torus, which eliminates the edge boundaries, making the comparisons easier. We can either directly test the spacing, or perform a two-level test. In the latter case, we can apply an AD test to the p-values found at the first-level test, or check the uniformity of the closeness of points [22].

5.1.4 Subsequences of random length

Lastly, we can also keep requesting numbers until an event happens. For example, the coupon collector's test creates a hypercube and uses vectors of generated numbers. We then split this hypercube into pieces of equal size and count how many numbers must be generated to have a number for each piece in the hypercube. We can then compare the number of numbers we had to generate to an expected frequency [22].

5.2 Test suites

All the tests described in Section 5.1 are divided into multiple test suites. An overview can be found in Table 5. The TestU01 library allows us to call on test battery, where a battery of tests is defined. For numbers within the range $U[0, 1]$, we can use the Crush test battery. There are three Crush test batteries, each with different intensity. The SmallCrush is the least stringent of the Crush test battery. It is comprised of 10 statistical tests, yielding 15 p-values. If a generator passes this test battery adequately, we can apply a more intense test battery, the Crush test battery. This test battery is comprised of 96 tests, yielding 144 p-values. If the generator still passes most of these tests, it can move on to the BigCrush test battery. This is the most encompassing test battery, having 106 tests and generating 160 p-values [22]. Lastly, we have an implementation of the DIEHARD test battery. This test battery is less stringent than the Crush test battery, but was considered a benchmark test battery before the TestU01 library.

Table 5: Statistical Test and how often it is featured in a test battery.

Test	Section	SmallCrush	Crush	BigCrush	DIEHARD
Birthday Spacings	5.1.3	1	7	9	10
Collision	5.1.3	1	–	–	–
Gap	5.1.1	1	4	4	–
Poker	5.1.2	1	4	4	–
Coupon Collector	5.1.4	1	4	4	–
Max of T	5.1.2	1	4	4	–
Weight Distribution	5.1.1	1	4	6	–
Matrix Rank	5.1.1	1	6	6	1
Hamming Independence	5.1.1	1	6	6	1
Random Walk	5.1.1	1	6	6	–
Serial Over	5.1.3	–	2	2	–
Collision Over	5.1.3	–	8	10	79
Close Pairs	5.1.3	–	4	4	2
Close Pairs Bit Match	5.1.3	–	2	–	–
Multinomial Bits Over	5.1.3	–	–	–	1
Run	5.1.1	–	2	2	1
Permutation	5.1.2	–	2	4	–
Collision Permutation	5.1.3	–	2	2	–
Sample Products	5.1.2	–	2	3	–
Sample Mean	5.1.2	–	1	2	–
Sample Correlation	5.1.2	–	1	2	–
Appearance Spacing	5.1.2	–	2	2	–
Sum Collector	5.1.1	–	1	1	–
Savir	5.1.1	–	1	1	1
Greatest common denominator	5.1.2	–	2	1	–
Linear Complexity	5.1.1	–	1	2	–
Lempel Ziv	5.1.1	–	1	2	–
Fourier	5.1.1	–	2	2	–
Longest Head Run	5.1.1	–	2	2	–
Periods in Strings	5.1.1	–	2	2	–
Hamming Weight	5.1.1	–	2	2	–
Hamming Correlation	5.1.1	–	3	3	–
Auto Correlation	5.1.1	–	4	4	–

6 Experiments

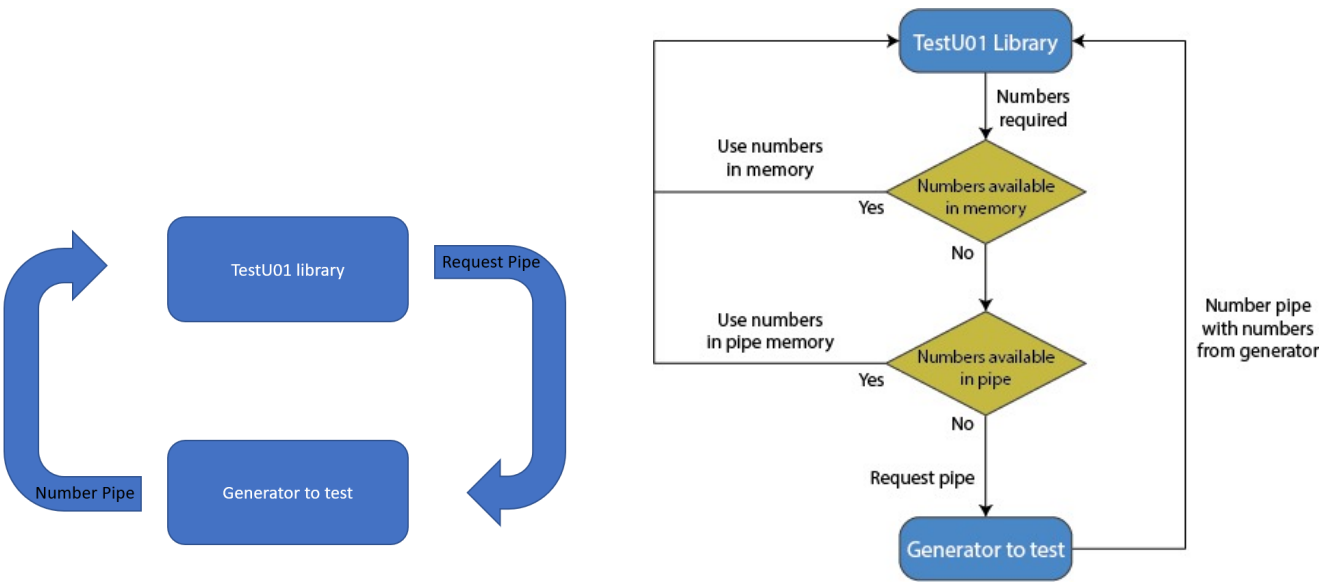
6.1 Approach

Since we want to test a plethora of programming languages, we need a way to transfer the generated numbers from the programming languages to the TestU01 C library, to allow us to compute the p-values required for our research on the generators. While this could be done via plain text files,

the number of required numbers is substantial: up to 2^{38} numbers are consumed by the BigCrush test suite. An improved solution for connecting the test suite to a generator of another programming language would be to use pipes. This allows us to connect the `stdin` of TestU01 to the `stdout` of another program. The language to test can continuously write a new number of its sequence to the `stdout`. As this method allows us to flexibly connect generators, this will be the approach we take.

To test a programming language's generator, we need to connect it to the TestU01 library. As mentioned before, we will use pipes to direct the output to the library. We will send a string representation from the programming language to the library using one pipe, and request new numbers from the programming languages using another pipe. These are named pipes, which can be instantiated on a Linux operating system using the `mkfifo` command. The pipes will continue to exist after a program termination or failure this way, and allow for more flexibility and a possibility of debugging if required. Figure 4a shows us the connection of the generator to test and the TestU01 library via named pipes.

Within the TestU01 library, we define a function that manages communication and transfers new numbers to the library. This will be done via polling. We check if there are still numbers available within our memory. If so, we consume one of the numbers and send it to the library. If we have none left in memory, we can check if our pipe still contains some numbers. If this is the case, we read numbers out of memory, convert them and store them into our memory. If none are left, a request is sent through the pipe to the programming language. The programming languages receive this request and will generate and write random numbers to the library via the other pipe. A flowchart of this can be seen in Figure 4b.



(a) Pipe layout (b) Flow chart

Figure 4: Overview of the TestU01 connection with the generator to test, using pipes.

In terms of hardware, several devices have been used to test the programming languages. One is a virtual machine with Ubuntu on a Windows host machine. Its CPU is an Intel Core i8-7700HQ 2.80GHz with 8 GB of RAM. The second device has an Intel Core i5-4590, 3,3 GHz (3,7 GHz Turbo Boost) CPU, with 8GB of RAM. It was run using the Windows subsystem for Linux, also known as WSL.

Several programming languages have been selected, which are shown in Table 6. Most of these programming languages have not gotten any previous assessment by the TestU01 library, except for Java, Mathematica, R and Matlab. This list tries to encompass a diverse set of programming languages. There are some older languages such as Fortran or C, as well as more recently developed languages such as Rust. This list also has many languages that have simulation and mathematical modelling as their core features, such as Matlab and Julia. Others are more commonly used for real-world applications, such as C++ and PHP.
Most programming languages have a separate method for generating random numbers within $U[0, 1]$, except for C, C++ and Bash. These only provide random numbers within the integer domain $\mathbb{Z}$, and only between 0 and the RAND_MAX variable, which is 0x7fff. To convert this to $U[0, 1]$, we can divide our generated integer by RAND_MAX. This distinction is also shown in Table 6, together with the method considered to be the default generator for that programming language.

Table 6: Overview of all tested languages, a brief description, the method considered default and required operations to transform its number to $U[0, 1]$

Language	Description	Method	Manipulation
C	a general-purpose language, designed to be easily translated into machine code [18]	`rand()`	divide by RAND_MAX
C++	a general-purpose language, allowing low-level programming and classes [38]	`rand()`	divide by RAND_MAX
Python	an interpreted high-level general-purpose programming language [42]	`random()`	–
NumPy library	a package within Python usually used for scientific computing [41]	`random()`	–
C#	a general-purpose, multi-paradigm programming language developed by Microsoft [1]	`random()`	–
Fortran	a general-purpose, compiled imperative programming language traditionally used for scientific computing [7]	`RANDOM_NUMBER(COST)`	–
Java	a high-level, general-purpose programming language [6]	`random.nextDouble()`	–
PHP	a popular general-purpose scripting language that is especially suited to web development [23]	`random_0_1()`	–
R	programming language for statistical computing [43]	`runif(1)`	–
Bash	GNU project shell for shell scripting [14]	`$RANDOM`	divide by RAND_MAX
Rust	A high-performance language for memory and power limited applications [40]	`rng.gen()`	–
COCO BBOB	a platform for comparing heuristic optimizers [15]	`uniform()`	–
COCO BBOB IOH	experimenter for iterative optimization heuristics [12]	`uniform()`	–
Octave	Mathematics-oriented programming language for modelling [46]	`rand()`	–
Julia	A programming language for AI, data science, and modelling [8]	`rand()`	–
Mathematica	Language by Wolfram for modern technical computing [44]	`Random[]`	–
Matlab	A mathematically oriented programming language, mainly used in engineering [30]	`rand()`	–
Javascript	Javascript is a just-in-time interpreted programming language used in browsers [32]	`Math.Random()`	–
Visual Basic	object-oriented programmin language developed by Microsoft [5]	rand()	–

6.2 Results

6.2.1 Seed results

As we try to find defects within a generator, seed selection is also important. The original publication [21] on Testu01 suite concluded based on empirical evidence that if a generator fails on a particular seed, it will fail on almost every seed. It is difficult to objectively use a neutral seed (a seed that does not greatly increase or decrease period potential) for each generator. This is especially true for the languages that have no clear description of its generator in the documentation. Since we want to check if the application of a neutral seed matters, we tested the use of different seeds and their impact on finding defect in the generator. We tested this with a standard LCG with the same modulus and multiplicant, but different seeds. In order to test this, we subjected the LCG with different seeds to the SmallCrush, and examined if we would observe different tests that would fail. The results can be found in Table 7.

Table 7: Reported number of failures of tests within SmallCrush on a LCG with 50 different seeds

			Test					
	Total over 50 runs	Average per run	1 [1]	2 [2]	6 [3]	8 [4]	10 [5]	8 & 10
Reported p-values	153	3.06	50	50	50	1	2	0
Suspect p-values	3	0.06	0	0	0	1	2	0

[1] Birthday Spacings Test
[2] Collision Test
[3] Max of T Test
[4] Matrix Rank Test
[5] Random Walk Test

As we can see in Table 7, choosing an arbitrary seed could result into one more test having a suspect p-value. We never see the combination of test 8 and 10 being reported, as shown in Table 7. However, the extra reported p-values are suspect p-values, with $10^{-10} < p < 10^{-4}$ or $1 - 10^{-10} > p > 1 - 10^{-4}$, while the p-values from tests 1, 2 and 6 are more extreme, which indicate defects within the generator. For the following experiments, seeding is not relevant for detecting our failures as of now and we can use one seed for all experiments, if we are allowed to supply one.

6.2.2 Generator results

Now that we have confirmed we can use an arbitrary seed for our experiment, we can test each languages generator. In Table 8, we can see the languages that were subjected to testing and the number of tests that they failed. We distinguish between a suspect p-value, where the p-value is in a grey zone where we cannot definitively conclude a failed test ($10^{-10} < p < 10^{-4}$ or $1 - 10^{-10} > p > 1 - 10^{-4}$), and a failure p-value. The failure p-value clearly indicates that a test has been failed by a generator, having $p < 10^{-10}$ or $p > 1 - 10^{10}$. The total number of both failed and suspect tests per battery is reported as the first value within a column, meanwhile the suspect

Table 8: Number of suspect and failure p-values generated when applying a test suite to a programming language, within brackets the suspect number of p-values is shown.

Language	Type	Version	SmallCrush	Crush	BigCrush	DIEHARD	Version[6]	SmallCrush	Crush	BigCrush
			Results in this thesis					Results in [21]		
Julia	MT	1.0.4	0	1	1	0	–			
Matlab	MT	R2021a	0	1	1	0	7	0	5	9 (1)
NumPy	MT	3.6.9	0	2(1)	2 (1)	0	–			
Octave	MT	6.2.0	0	0	1	0	–			
Python	MT	3.6.9	0	1	1	1 (1)	–			
PHP	MT	7.2	0	2 (1)	2	1	–			
R	MT	4.0.5	0	0	2	1	2.4.1	3 (1)	44 (4)	–
Rust	MT	1.53.0	0	0	2	0	–			
Visual Basic[7]	MT	16.9	0	1	2	0	–			
Bash	LCG	4.4.20	15	–	–	126	–			
C	LCG	7.5.0	15	–	–	126	–			
C++	LCG	7.5.0	15	–	–	126	–			
COCO BBOB [8]	LCG	X	11 (3)	–	–	–	–			
COCO BBOB	LCG	X	1	9	9	0	–			
Java	LCG	8	1	12 (3)	22 (1)	8 (4)	SE 6	1	12 (3)	22 (1)
C#	LFib[9]	8.0	2 (1)	11 (2)	11	2	–			
Fortran	XSR[10]	2018	0	0	0	0	–			
Mathematica	LSFR[11]	12.3	0	0	0	0	6.0	3 (2)	18 (3)	–
Javascript	NA[12]	16	0	0	1	0	–			
total number of tests in the suite			15	144	161	126		15	144	161

[6] The original publication [21] does not report the versions of languages. Versions provided here are roughly calculated by us based on the time of the publication of [21]

[7] Excel uses Visual Basic for its RANDOM() implementation

[8] COCO BBOB generator was altered to other function suite

[9] LFib: substract-with-borrow, variation of Lagged-Fibonacci [1]

[10] XSR: Xorshiro256** [3]

[11] LSFR: ExtendedCA, LSFR with 5 non-adjacent previously generated numbers [2]

[12] NA: there was no mention of the implemented method for Javascript

p-values are marked within brackets after the total, if any.

As we can see in Table 8, almost no suspect p-values are reported within SmallCrush. Only Java and C# have suspect p-values. Java fails the Birthday Spacings test, with $p < 10^{-300}$. The Birthday Spacings tests are described in Section 5.1.2. Vectors are formed, and we split a hypercube into k cells. We then count the collisions between spacings, which is compared to an expected collision distribution. LCGs fail this test often, as their spacings tend to be more regular than what would be expected [21]. We see the same, with the C# generator. The C# SmallCrush battery reports one more test, the Weight Distribution test, also described in Section 5.1.1. It is a suspect p-value, with $p = 1.2 \times 10^{-7}$, which is less severe.

The LCGs used in C, C++ and Bash fail all tests with either $p < 10^{-300}$ or $p > 1 - 10^{-300}$ [13], which indicate clear failure. These values are extreme due to the large sample size used in each test. They are therefore not subjected to heavier test batteries, as this would also result in total failure. This is because the tests used in Crush and BigCrush are more stringent than the tests used in SmallCrush. The same is done with the modified COCO BBOB, which has 11 p-values reported, of which 8 are failures. The other languages do not have any failures within SmallCrush.

When we look at the results on the Crush test battery in Table 8, we see that more languages have tests with suspect p-values. Python, PHP and Visual Basic, Matlab and Julia all fail the same test, the linear complexity test. The linear complexity test measures the linear complexity of a sequence. The linear complexity of a sequence is calculated by the TestU01 library with the Berkelenkamp-Massey algorithm [21], which tries to find a LSFR for a given sequence [34]. The Crush test batteries has two variations of this test, one on all of its bits and one with only its least significant bits. All only fail the test on the former variant, when there are no bits dropped. This does however only generate a suspect p-values.

PHP get a suspect p-value from another test within the Crush battery, the random walk test. The random walk test takes l bits from the uniform numbers, and uses each bit within that number to either add one or substract one. We start at zero and repeat the bit consumption until we have walked L steps, consuming L bits and ending the walk at integer x. If we repeat this process, we get a collection of integers X. We can store various properties of each walk, such as sign changes or the first time we land on integer y, and compare these properties to an expected distribution, to try and detect nonuniformness [21].

When we look at the Java, C#, and unmodified BBOB results within the Crush battery, we see that they both fail all variants of the birthday spacings test. They also generate suspect p-values for the linear complexity tests.

Finally, we can analyze the results for the BigCrush test battery. We see that few programming languages fail no tests. Only Fortran and Mathematica have no suspect p-values or p-values that indicate a failure. Javascript only has 1 failure, the linear complexity test. This also happens to all languages that have some implementation of the Mersenne Twister. Julia, Matlab, Octave and

[13] 64 bit double precision

Python only fail on the linear complexity test performed on the least significant bits. NumPy, PHP, R, Rust and Visual Basic also fail the linear complexity test variant where it performs the test on all available bits.
As for Java, BBOB, and C#, we see that they continue to fail the birthday spacings tests. These are slightly more stringent than the variants used in the Crush battery, which they also failed. Java also fails other tests, the collision over test and the random walk test. The collision over test creates overlapping vectors as described in Figure 3, and splits the hypercube into segments. It then counts the amount of collisions per segments, and compares these to an expected distribution.

6.2.3 P-value distribution

Each number generated is only used in one test. L'Ecuyer and Simard report [21] that rerunning tests typically does not improve efficiency. This is because a generator typically has the same patterns we try to detect across all subsequences, if we use a reasonably long subsequence [21]. The number of numbers used per test varies both among test batteries and individual tests within a battery, from 20000 for a binary matrix rank test to 10^9 numbers for a typical test within BigCrush. These large sample sizes allow for very small p-values, which in turn allow for decisive rejection in a test. For each test with a set of parameters, we only run that test once, $N = 1$. In Table 9, a summary can be found on how often a test reports a suspect p-value of a p-value that indicates failure of the generator.
Many tests do not show us any deficiencies across all tested languages and batteries (apart from C++, C, and Bash which fail all tests on SmallCrush). We only have five tests that report suspect p-values at all:

- Birthday Spacings Test
- Weight Distribution Test
- Random Walk Test
- Linear Complexity
- Collision Over

While some languages fail some tests quite severely, most languages have been able to withstand a majority of the tests, even within BigCrush, the most extensive test battery. The tests generally fail on linear complexity test and matrix rank test. Since the generators that fail these tests are related to Mersenne Twister, we could argue that these generators are a variation of the algorithm, which seems to be widely adopted by language developers. This is also visible within the documentation of these programming languages.

Table 9: Statistical tests and their number of suspect p-values reported across 19 considered languages, excluding C, C++ and Bash (since they failed each test on SmallCrush). Here NA denotes a test not being included in the test battery.

	suspect or failure p-values			
Test	SmallCrush	Crush	BigCrush	DIEHARD
Birthday Spacings	4	27	28	1
Collision	1	NA	NA	NA
Gap	1	0	0	NA
Poker	1	0	0	NA
Coupon Collector	1	0	0	NA
Max of T	2	0	0	NA
Weight Distribution	1	0	0	NA
Matrix Rank	0	0	0	1
Hamming Independence	0	0	0	0
Random Walk	3	1	5	NA
Serial Over	NA	0	0	NA
Collision Over	NA	0	4	7
Close Pairs	NA	0	0	0
Close Pairs Bit Match	NA	0	NA	NA
Multinomial Bits Over	NA	NA	NA	1
Run	NA	0	0	0
Permutation	NA	0	0	NA
Collision Permutation	NA	0	0	NA
Sample Products	NA	0	0	NA
Sample Mean	NA	0	0	NA
Sample Correlation	NA	0	0	NA
Appearance Spacing	NA	0	0	NA
Sum Collector	NA	0	0	NA
Savir	NA	0	0	0
Greatest common denominator	NA	0	0	NA
Linear complexity	NA	13	18	NA
Lempel Ziv	NA	0	0	NA
Fourier	NA	0	0	NA
Longest Head Run	NA	0	0	NA
Periods in Strings	NA	0	0	NA
Hamming Weight	NA	0	0	NA
Hamming Correlation	NA	0	0	NA
Auto Correlation	NA	0	0	NA

Table 9 shows that most observed programming languages supply generators that are resilient to most tests. When we check the p-values that get reported by these tests, we see that most failures relate to LCG or LFib use, which are generators with known deficiencies. Mersenne Twister, which is more sophisticated, does perform significantly better. The linear complexity test and the birthday spacings test have reported failure p-values instead for suspect p-values. Failure p-values show

definitive defects within generators, whereas suspect p-values could also happen to be a Type I error, where we wrongly reject H_0. Since we run thousands of tests across all tested languages, we are bound to have some suspect p-values that are Type I errors.

Some tests, like the gap or poker test, do never fail within our benchmark, aside from the precision issue seen in C, C++ and Bash, but these are still important to ensure developers do not make any trade-offs within the generator while trying to avoid failing other tests. For example, a test should never fail global uniformity, even if it fixes the birthday spacing failure. Even the most basic generators should pass this reqruirement and all generators that are default within programming languages should adhere to these principles.

6.3 Discussion

6.3.1 Pipes

The pipe-based system used in this thesis is not optimal. Reading from pipes when there is no input sends the programming languages that are tested an End-of-file (`EOF`), which blocks all communication with the programming language. This halts the testing entirely, as no new numbers are transferred for TestU01 to process. Javascript and Matlab for example do not allow reading from the `stdin` after receiving an EOF. Allowing the TestU01 to read multiple numbers at once and store these could also greatly improve efficiency, as well as optimizing writing to the `stdout` for each programming language. Some languages, such as Excel, are not directly command-line based. This makes it difficult to connect these to the TestU01 library directly. Testing, therefore, happens indirectly via Visual Basic. The solution to only make the pipes contain one number at most and regulate this to not consume an EOF is suboptimal in terms of performance, but does allow us to test these languages.

6.3.2 Precision

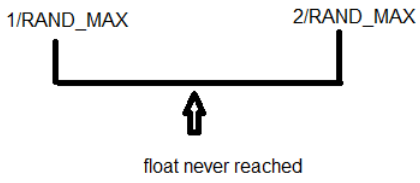


Figure 5: Some numbers in $U[0, 1]$ are never reached with the PRNG in C, C++ and Bash

C, C++ and Bash fail every test, which is related to the limited precision of the numbers generated. This is because we have to divide by the RAND_MAX, allowing us only limited precision, as shown in Figure 5. Most tests require these to be 30-bit or above, while C, C++ and Bash do not have this precision.

6.3.3 Requirements

We want to be able to make a decision on which generator we can use when we require random numbers for a certain task. If we do not care about reproducability and do not consume a lot of numbers per second, we can opt for a true random number generator (TRNG), a generator that does not rely on mathematical formulae. This usually involves using /dev/random on Linux systems, or some other way to generate entropy for creating these numbers. For lottery draws, sampling, games and security, a TRNG is preferred [16]. In case we want to simulate or model something, reproducability is vital, which seeding allows us to do within a PRNG. We also need to possibly consume many random numbers within these simulations. In these cases, a PRNG is preferred [16]. For generating procedural noise, we also want a PRNG [33]. In the case we want to write unit tests (small tests that test components of code) for our code that involve some sort of random number generation, a PRNG is also preferred. Seeding this PRNG allows us to reproduce unit tests that fail and improve the code that is tested.

If we do use a PRNG, we need to keep the seed state into consideration. If we want to for example shuffle a deck of N cards, we have $N!$ possible deck permutations. If a PRNG seed does not allow for $N!$ bits, we cannot achieve every possible permutation [33]. If this is vital for the application, we should not pick this generator.

So all of the generators would be able to, for example, perform 10 dice rolls. However, if we want to use heuristic optimisation methods, we could see an increase in performance if we select the generator that has the fewest failures within BigCrush.

7 Conclusions and Further Research

7.1 Conclusions

In this study, we have benchmarked generators from commonly used programming languages. While some deficiencies can be found in most programming languages, they are relatively small problems. Using this generators should not create problems.

If we want to avoid known problems with generators that are considered default within a programming language, a solution would be to look for a generator within the programming language other than the default that is already implemented. Most programming languages have libraries that implement thoroughly tested PRGN algorithms. If both of these options are undesired, consider switching to another language that does allow for a proper implementation of its random number generator.

Some cautiousness is advised when using the Java and C# generators, as well as the C, C++ and Bash generators, as they show some deficiencies. These major deficiencies should be fixed by their language or library developers before they should be considered. An explanation for the worse performance when compared to generators used in other languages is that these languages generators implemented their algorithms earlier than others. These algorithms have usually been found outdated, as the requirements for simulation have increased, while the algorithms have not

improved. Changing these generators could not be a priority for them, since users mostly are not aware of the potential deficiencies these generators have and backwards compatibility is also required for these commonly used functions.

7.1.1 Mersenne Twister

As for the Mersenne Twister, its period is substantial contributor to its popularity, along with its robustness when subjected to statistical tests, which we have confirmed with our experiments. These properties make it a widely adopted method for implementing pseudorandom number generators. It could be upgraded to SFMT for additional resilience, but it is still better than most algorithms used and suffices for a general-purpose generator. Xorshiro is also recommendable, as it passes all of the BigCrush tests as the Fortran generator. When a user wants to use a PRNG, the user could be expected to review the purpose they want to use this generator for. If this requires strong statistical properties, it would be wise to inform him- or herself about the properties of the generator it uses. But although this is true, using known flawed generators as the default within a programming language could result in problems for users. A clear example of an application failing was shown in Section 2.

There seems to be a difference between using the random number generator supplied by the NumPy library and the default random number generator of Python. While this does not seems to lead to major differences, it seems to be better to use Python over NumPy, although both will perform decent on everyday applications and can be used for simulation.

Some p-values that fail on NumPy and PHP are suspect p-values that were close to not being reported, and do not get reported when BigCrush is applied. These can be considered false positives, where we do get a suspect p-value while there is no real problem. We would expect some values to come close to suspect when testing many generators, since we run many tests and generate a substantial number of p-values. As long as these values stay within the suspect range and do not persist in heavier testing, they can be regarded as warnings rather than problems. TestU01 would allow us to write custom parameters for tests, which would allow us to check if these suspect p-values persist.

All of the algorithms that have implemented some variation of Mersenne Twister are adequate for everyday use. They do not show major deficiencies, even when subjected to stringent tests. Using the PRNGs of these programming languages should not lead to decreases in performances or the failure of applications. They are only susceptible to linear complexity tests, which would only have major consequence for security applications. None of these PRNGs should ever be used for these purposes, as this would pose major cryptographic threats.

7.1.2 LCG

C, C++, and Bash are not performing well in terms of resolution and because of their use of the older LCG algorithm, which has previously been shown to underperform by the TestU01 test suite. Replacing these with more modern solutions would increase the resilience against statistical testing. The Java generator fails the birthday spacing tests, which is common for LCGs to fail, as its spacing

is directly linked to its algorithm. For 10 years, nothing has changed and it should be updated to a more resilient algorithm. There are several algorithms that are more resilient and faster than the one currently used. Until improvements to these implementations have been made, refraining from using these would be better. These could still be used for small-scale number generation. However, for larger applications that require uniformness to function properly it would be unwise to use these.

BBOB, in its modified version, does fail severely on a majority of the tests in the SmallCrush battery. Since we have another version available, it would be better to use the unmodified version, which does perform significantly better. It only fails on the birthday spacings tests in both Crush and BigCrush, making it a decent variant to use in simulation work. However, if a more robust method is available, it should be swapped with something that does pass the birthday spacing tests, as it shows a too evenly distributed generator. For simulating true randomness, we would expect more collisions to happen than we see with this method.

7.1.3 Other generators

The C# SWB also has its flaws, since it fails several tests. This problem can be mitigated by adopting another solution for generating random numbers, for example using a library that implements a more robust algorithm, such as xorshiro256**.

Fortran seems to pass these tests in each battery effortlessly, not having any suspect p-values or p-values that clearly indicate failure. Upon researching the documentation, it turned out that Fortran implements the xorshiro256** algorithm for its PRNG. This algorithm passes each test within BigCrush according to its publication paper [9].

Javascript has a resilient PRNG, which only fails one test within the BigCrush test battery. This PRNG could be considered among the best we have tested.

Mathematica has a rare implementation of its generator, the extendedCA. While an uncommon implementation usually would mean decreased performance due to unforeseen errors being introduced into the generator or limited testing, it is not the case in this instance. Mathematica performs without error in the TestU01 test suite. This PRNG could be used for demanding tasks, such as simulation or heuristic optimization methods.

7.1.4 Comparison to older measurements

It is also interesting to see how the languages generators that have been tested before in the TestU01 publication paper [21]. Many of the languages have since then adopted a new generators as their default, while some have kept their generator.
Java still seems to have the same issues today as in its testing in the TestU01 paper, indicating that nothing has changed yet. In September of 2021, a new Java update (Java JDK 17) is expected [37] to alter the random number generator framework, but it remains to be seen if the default generator is also improved.
R seems to have improved since being tested by the TestU01 original publication, as it switched

from a faulty multi-carry implementation to the more reliable Mersenne Twister as its default implementation. If others that have insufficient results would also switch to Mersenne Twister, they could see similar improvements.
As for Mathematica, it uses a robust, relatively new method for generating random numbers. This method is resilient according to our testing, passing all tests without any suspect p-values or p-values that indicate failure. This is a significant upgrade compared to the method it used in the TestU01 paper, where its substract-with-borrow would fail many tests and was not even submitted to the BigCrush test battery.
Matlab has also implemented a new PRNG, using the Mersenne Twister instead of its older method for generating random numbers. In this case it is also an upgrade, having only 1 failure p-value instead of 8 in the TestU01 paper.
In most cases, improvements have been made since the original paper tested these programming languages. Java is the only one that still has the same results, although this could change in the future when the new JDK version is released.

7.2 Further research

As for further research, other languages can also be subjected to the TestU01 test batteries. New languages are being developed continuously and these too need to provide proper generators to their users. We could also see new statistical tests being developed in the future, allowing us to expose faults within generators that could be problematic. The statistical tests that are used in TestU01 are more stringent than the DIEHARD tests. We could similarly see new tests being added within TestU01. Continuously testing default generators is important, because these generators are frequently used in everyday coding for applications with varying requirements on the quality of random sequences. It would be recommended for language developers to assess their implementation of a pseudorandom number generator routinely, and publish their results with their release notes.

As mentioned previously, this research focused on PRNGs. Many programs also supply a cryptographically secure pseudorandom number generator, which could be also be benchmarked in further research. Not only do they still need to adhere to the same requirements of uniformity as their non-cryptographic counterparts and could be subjected to the same statistical tests, they also need to satisfy additional requirements explained in Section 2. While the test batteries within TestU01 do not check for the cryptographic properties, they should still be able to pass all of the tests within TestU01, as these are necessary for its cryptographic properties as well [35]. A CPRNG needs to pass these, otherwise, patterns within the CPRNG can be discovered, which would be bad. This would violate one of its core principles, where a bit can be predicted by its preceding bits, if an adversary were to discover these. As mentioned by Mike Malone, using a CPRNG could be a valid alternative for a PRNG [24], although a user might have to sacrifice number generation speed.

Another use case of a PRNG, although used less often, is generating bits, rather than numbers within $U[0, 1]$. This might be preferred to generating within $U[0, 1]$ in an embedded system where we need bit generation, or for testing random number generators based on physical devices, as we consume less computation time and energy. TestU01 also has test batteries for this, the Rabbit batteries [21]. These allow us to do similar tests as the Crush libraries. These use significantly less data, around 2^{20} bits, and are file-based, rather than continuous. Further research could be done

to analyze the different bit generators that are used in programming languages via the Rabbit batteries. The focus for now was the testing of generators that generate within $U[0, 1]$, as these types of generators are commonly used for applications.

References

[1] C# docs - get started, tutorials, reference. — Microsoft Docs. URL: https://docs.microsoft.com/en-us/dotnet/csharp/.

[2] Random Number Generation—Wolfram Language Documentation. URL: https://reference.wolfram.com/language/tutorial/RandomNumberGeneration.html.

[3] RANDOM_NUMBER (The GNU Fortran Compiler). URL: https://gcc.gnu.org/onlinedocs/gfortran/RANDOM_005fNUMBER.html.

[4] RANDOM.ORG - Statistical Analysis. URL: https://www.random.org/analysis/.

[5] Visual Basic docs - get started, tutorials, reference. — Microsoft Docs. URL: https://docs.microsoft.com/en-us/dotnet/visual-basic/.

[6] What is Java? URL: https://www.java.com/nl/about/whatis_java.jsp.

[7] John Backus. Home - Fortran Programming Language. URL: https://fortran-lang.org/.

[8] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A Fast Dynamic Language for Technical Computing. 2012. URL: https://julialang.org/http://arxiv.org/abs/1209.5145.

[9] David Blackman and Sebastiano Vigna. Scrambled Linear Pseudorandom Number Generators. Technical report, Università degli Sstudi di Milano, Milan, 2019.

[10] Robert Brown, Dirk Eddelbuettel, and David Bauer. Robert G. Brown's General Tools Page, 2011. URL: http://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[11] Erick Cantu-Paz. On Random Numbers and the Performance of Genetic Algorithms. Technical Report 10, 2002. URL: www.random.org.

[12] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics. arXiv, 2018. URL: https://github.com/IOHprofiler/IOHexperimenter.

[13] Sonja Engmann and Denis Cousineau. Comparing distributions: the two-sample Anderson-Darling test as an alternative to the Kolmogorov-Smirnoff test. Journal of Applied Quantitative Methods, 6(3):1–17, 2011. URL: https://www.researchgate.net/publication/276918573.

[14] Brian Fox. Bash - GNU Project - Free Software Foundation, 2014. URL: http://www.gnu.org/software/bash/.

[15] French National Research Agency. Comparing Continuous Optimisers: COCO, 2013. URL: https://coco.gforge.inria.fr/http://coco.gforge.inria.fr/doku.php?id=algorithms.

[16] Mads Haahr. random.org: Introduction to Randomness and Random Numbers. Statistics, (June):1–4, 1999. URL: https://www.random.org/randomness/.

[17] David Jones. Good Practice in (Pseudo) Random Number Generation for Bioinformatics Applications. Technical report, UCL Bioinformatics Group, Londen, 2010. URL: http://www.iro.umontreal.ca/~lecuyer/myftp/papers/testu01.pdf.

[18] B W Kernighan and D M Ritchie. The C Programming Language. 1988.

[19] Donald E. Knuth. The art of computer programming. Stanford university, Reading, 2 edition, 1981.

[20] Pierre L'Ecuyer and Francois Panneton. A new class of linear feedback shift register generators. Technical report, Université de Montréal, Orlando, FL, USA, 2000.

[21] Pierre L'Ecuyer and Richard Simard. TestU01: A C library for empirical testing of random number generators. ACM Transactions on Mathematical Software, 33(4):1–40, 2007. doi:10.1145/1268776.1268777.

[22] Pierre L'Ecuyer and Richard Simard. TestU01 A Software Library in ANSI C for Empirical Testing of Random Number Generators User guide, compact version. Technical report, Université de Montréal, Montréal, 2013. URL: http://www.iro.umontreal.ca/~lecuyer/.

[23] Rasmus Lerdorf. PHP: Hypertext Preprocessor. In Encyclopedia of Systems Biology, pages 1704–1704. 2013. URL: https://www.php.net/, doi:10.1007/978-1-4419-9863-7{_}101153.

[24] Mike Malone. TIFU by using Math.random()., 11 2015. URL: https://medium.com/@betable/tifu-by-using-math-random-f1c308c4fd9d.

[25] George Marsaglia. Random number generators, 5 2003. URL: http://digitalcommons.wayne.edu/jmasmhttp://digitalcommons.wayne.edu/jmasm/vol2/iss1/2http://digitalcommons.wayne.edu/jmasm/vol2/iss1/2, doi:10.22237/jmasm/1051747320.

[26] George Marsaglia. Xorshift RNGs. Technical report, The Florida State University, 2003. doi:10.18637/jss.v008.i14.

[27] George Marsaglia and Arif Zaman. Toward a universal Random number generator.pdf. Technical report, The florida State University, Tahllahassee, 1987. URL: https://web.archive.org/web/20100610050921/http://stat.fsu.edu/techreports/M766.pdf.

[28] Michael Mascagni, Steven A Cuccaro, Daniel V Pryor, and M L Robinson. A fast, high quality, and reproducable parallel lagged-fibonacci psuedorandom number generator. Technical report, Supercomputing Research Center, I.D.A, Bowie, 1994.

[29] Michael Mascagni and Ashok Srinivasan. Algorithm 806: SPRNG. ACM Transactions on Mathematical Software, 26(3):436–461, 9 2000. URL: https://dl.acm.org/doi/10.1145/358407.358427, doi:10.1145/358407.358427.

[30] Mathworks. MATLAB - Mathworks - MATLAB & Simulink, 2016. URL: https://nl.mathworks.com/products/matlab.htmlhttps://nl.mathworks.com/products/matlab.html%0Ahttp://www.mathworks.com/products/matlab/.

[31] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. ACM Transactions on Modeling and Computer Simulation, 8(1):3–30, 1998. doi:10.1145/272991.272995.

[32] MDB web docs. JavaScript - — MDN. URL: https://developer.mozilla.org/en-US/docs/Web/javascripthttps://developer.mozilla.org/es/docs/Web/JavaScript%0Ahttps://developer.mozilla.org/en-US/docs/Web/JavaScript%0Ahttps://developer.mozilla.org/en-US/docs/Web/JavaScript%0Ahttps://developer.mozilla.or.

[33] Peter Occil. Random Number Generator Recommendations for Applications. 2018. URL: https://peteroupc.github.io/random.html#High_Quality_RNGs_Requirements.

[34] J A Reeds and N J A Sloane. Shift-register synthesis (modulo m). Society for Industrial and Applied Mathematics, 14(3), 1985.

[35] Andrew Rukhin, Juan Soto, and James Nechvatal. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications. (April), 2010.

[36] Mutsuo Saito and Makoto Matsumoto. SIMD-oriented fast mersenne twister: A 128-bit pseudorandom number generator. Monte Carlo and Quasi-Monte Carlo Methods 2006, pages 607–622, 2008. URL: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html., doi:10.1007/978-3-540-74496-2{_}36.

[37] Guy Steel. JEP 356: Enhanced Pseudo-Random Number Generators, 2021. URL: https://openjdk.java.net/jeps/356.

[38] Bjarne Stroustrup. C++.org. URL: http://www.cplusplus.org/.

[39] Ronald J. Tallarida and Rodney B. Murray. Chi-Square Test. In Manual of Pharmacologic Calculations, pages 140–142. Springer New York, New York, NY, 1987. URL: http://link.springer.com/10.1007/978-1-4612-4974-0_43, doi:10.1007/978-1-4612-4974-0{_}43.

[40] The Rust Team. Rust Programming Language, 2020. URL: https://www.rust-lang.org/.

[41] Stéfan Van Der Walt, Gael Varoquaux, and Saclay France. The NumPy array: a structure for efficient numerical computation. 2011.

[42] Guido van Rossum. About Python™ — Python.org. URL: https://www.python.org/about/.

[43] Adam Wilson and Natalia Norden. The R Project for Statistical Computing, 2015. URL: https://www.r-project.org/.

[44] Wolfram version 11.3. Wolfram Mathematica: Modern Technical Computing, 2019. URL: https://www.wolfram.com/mathematica/https://www.wolfram.com/mathematica/index.html.en?footer=lang%0Ahttps://www.wolfram.com/mathematica/.

[45] Andrew Yao. Theory and application of Trapdoor functions. Technical report, University of California, Berkely, 1982.

[46] Washizawa Yoshikazu. GNU Octave. The journal of the Institute of Image Information and Television Engineers, 65(5):683–686, 2011. URL: https://www.gnu.org/software/octave/http://ci.nii.ac.jp/naid/110009669142/.

[47] Robert M. Ziff. Four-tap shift-register-sequence random-number generators. Technical report, University of Michigan, 2018.