Evaluating the way in which the languages Python and JavaScript generate random numbers.

Computer Science Extended essay.

Examination session: May 2023

Word count: 4000

Contents

Introduction 3
Pseudo Random Number Generators (PRNGs) 4
Testing Random Number Generators. 5
Chi-Square test 6
PractRand 8
Python's PNRG 8
Mersenne Twister 19937 9
Testing Python's RNG 10
Chi-squared test 10
PractRand 11
JavaScript's PRNG 12
Xorshift128+ 12
Testing JavaScript's RNG 13
Chi-squared test 13
PractRand 15
Comparison 15
Chi-squared test 15
PractRand 16
Conclusion 16
"Best" RNG 16
Personal verdict 17
Appendix A: Practrand Standard Test Battery 19
Bibliography 21

Introduction

Randomness is an ambiguous concept. In theory one can generally define something as random if its outcome is unpredictable, rolling a number on a die is considered a random event since there is no practical way to predict its outcome. However, the key part in that definition is practical, as if one was to have the precise information about all the parameters that played a part in the die drop (height, force, angle, air resistance…) the outcome of this roll could be precisely calculated using physics formulas. Now the commonly deemed "random" die roll doesn't seem random anymore as it follows a deterministic system. Nowadays there are events which scientists consider to be truly random such as radioactive decay, which according to the quantum mechanical laws we understand has an inherent uncertainty which makes it random (Science direct, n.d.). Nevertheless, the possibility that science hasn't advanced enough to predict these events still exists. In fact, if someone had all of the data for any possible "random event" he might find that the entire universe is deterministic, and that there are no "chaotic events" at all, but this goes into a bigger debate that's outside the focus of this essay. Knowing this, one can conclude that randomness is a measure of the observer's ignorance rather than an inherent quality of an event.

Despite randomness being such an erratic concept, it is exactly this quality which makes it so important and valuable to society. Randomness is used daily to a great extent, from using it for leisure in gambling, to security in generating cryptographically strong passwords, passing through its use in simulating real life events in gaming, or something as simple as choosing who will be the first group to present their project in class. There is no denying that there is a need for us to have access to "random events" in the sense that the observer will not be able to predict their outcome.

As previously established, most events outside the quantum realm are considered to be deterministic, and computers are no exception to this rule. By definition, a computer is a programmable device capable of carrying out multiple processes involving the storage and manipulation of data (Merriam-Webster, n.d.), the key part of this definition is the word programmable, to put it simply a computer receives precise orders

and executes them, so how does something that does exactly what it is told to do possibly do something unpredictable? In a similar fashion to the die, it cannot.

In my years of programming, I've had to use random number generators (RNGs) extensively, the main languages which I have used in general, and consequentially for the purpose of generating random numbers, have been Python and JavaScript. The purpose of this extended essay is to dive into how computers manage to randomly generate numbers, and to evaluate between the programming languages of Python and JavaScript which one does the job best.

Pseudo Random Number Generators (PRNGs)

As previously mentioned, computers are deterministic machines, this means that they obtain an input, and produce and output. A Pseudo-Random Number Generator (PRNG) is the process by which a computer generates a number by obtaining an initial value called a "seed" and inputs this value into an algorithm which will make the number into a new one by putting it through a series of mathematical equations until the algorithm outputs an unrecognizable "random" number (Khan Academy, n.d.). The seed can usually be obtained by drawing values from many sources such as the time at which the process occurs, the input of a user, or even the time between the keystrokes someone typing in a keyboard.

The "problem" with these Random Number Generators (RNGs), is that by the definition of what an algorithm is, the process is completely deterministic, because of this the randomness of the process depends on the seed.

These types of RNGs are good for most intents and purposes, in fact, the RNGs which will be evaluated in this essay are both PRNGs. However, it is worth mentioning that there is such a thing as a True Random Number Generator (TRNG), these generators are not digital, and require some hardware as their way to obtain their seed is by recreating some real-world phenomena which is considered to be random.

“HotBits” is an example of a TRNG, which obtains its random numbers from the aforementioned radioactive decay (Walker, 1996).

Testing Random Number Generators.

Leaving the philosophical question of whether randomness is achievable behind, when testing RNGs no matter their nature, there are two main focuses to measure.

- The distribution of outcomes.
- The correlation between the outcomes.

The first focus refers to how the frequency of the digits is spread. This is really important as a truly random RNG, in order to be unpredictable, should end up with a uniform spread of all numbers. By analyzing the distribution of outputs an RNG gives it is possible to check for any potential bias the RNG might have.

For this study, the main method used to evaluate this distribution of outcomes will be a Chi-square “goodness of fit test”.

The second focus refers to the correlation that might exist between the outcomes. RNGs are algorithms which use equations to output numbers based on a seed, because of this, they all follow a pattern in how they output results. What sets apart a good RNG from a bad one, is how unrecognizable the pattern is.

There are many ways in which this pattern may express itself, for example an RNG’s period is how long it takes for the RNG to repeat its sequence of outputs from the very beginning, but there usually are other less obvious correlations which can be observed within an RNG’s output in a short range.

For this study, the PractRand testing suite will be used to check for correlations in the data stream each RNG outputs.

Chi-Square test

A Chi-square (X^2) goodness of fit test measures how well a statistical model fits a set of data. However, it can be repurposed to test an RNG's distribution of outcomes.

In order to test the RNGs with this method, each one will be set to generate a number of integers between 0 and 9, after this, the frequency of each integer will be counted. In this way, the Chi-square test will be used by making the set of data be the observed frequencies for each randomly generated integer, and the statistical model be that every frequency will be equally distributed.

In this test a Chi-square statistic will be obtained by using the Chi-square formula.

$$X^2 = \sum_{k=1}^{n} \frac{(O_k - E_k)^2}{E_k}$$

Where *O* represents the observed frequency of an integer and *E* its expected frequency (JMP, n.d.).

Since there are 10 possible integers to be generated n = 10.

This test statistic represents the difference between the expected and actual frequencies observed, so for the purpose of comparing the RNGs, the lower the statistic the better. However, since this is a statistical test, it is necessary to check if the statistic passes the test.

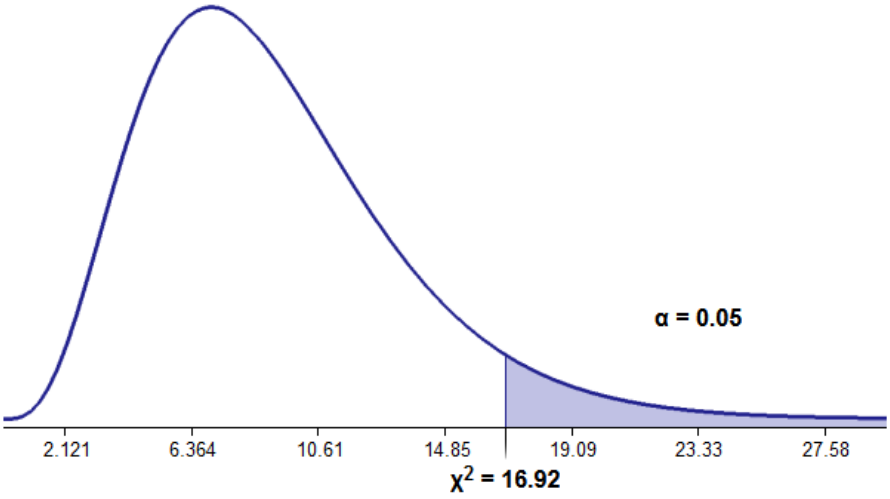
For this purpose, a critical value will be calculated, and later compared to the test statistic, this critical value is obtained from the Chi-square distribution. In order to obtain this value, it is necessary to set a significance level ($\boldsymbol{\alpha}$) for the test, this will represent the risk of the data not actually fitting the model, in this way, the lower the significance level, the stronger that the evidence of the data fitting our model will need to be, while a higher significance level will allow some breathing room for the data to not perfectly fit the model. For this test a 5% risk will be allowed, low enough so that if the RNG passes the test, confidence in its outcomes being equally distributed can be attained, but high enough so that it will allow some room for the distribution to not be an exact spread due to probability. Thus $\boldsymbol{\alpha}$ = 0.05.

Once the significance level is selected, the next step is to look at the Chi-square probability density function (pdf). This distribution is plotted by the following function.

$$f(x|k) = \frac{1}{2^{k/2}\Gamma(k/2)} x^{k/2-1} e^{-x/2}$$

This function takes *k* as a parameter and contains a gamma function (Γ) which takes *k*/2 as its *x* value. Where *k* represents the degrees of freedom of the test, this is obtained by taking the number of integers which can be generated, which is 10, and subtracting 1, so *k* = 9. (Weisstein, n.d.)

Once this function is plotted, the critical value will be the cutoff point for the right tail of the function, so that the area bellow it equals the significance level. Technology can be used to find that this critical value is ≃ 16.92.



(Johnston, n.d.)

Thus, in order to pass the Chi-square goodness of fit test with a 5% margin of error, the RNG's Chi-square statistics will need to be lower than 16.92.

Finally, as with any goodness-of-fit-test, it is important to obtain a p-value to support the conclusion. A p-value is a measure of the evidence against the null hypothesis in a statistical test, in this case the null hypothesis being that the distribution of values will not be uniform. The p-value is compared to the significance level, and if it is greater the null hypothesis is likely to be true. So a p-value lower than 0.05 would suggest that any uniformity in the distribution has happened due to chance as there isn't enough data

to prove that the RNG is random, while a p-value higher than 0.05 would provide evidence for the RNG having a uniform distribution due to its random nature.

The p-value is calculated using the same probability density function used for the critical value, though instead of calculating the chi-squared value, it is calculated by cutting of the function at the resultant chi-squared value and calculating the area of the right tail end. Because of this definition, if the chi-squared value is below the critical value the p-value will always be above the significance level, however in this situation it serves to see how strong the evidence is to either reject or accept the null hypothesis. (Mindrila & Balentyne, n.d.)

PractRand

PractRand, also known as "Practically Random" is a C++ library of pseudo-random number generators, and statistical tests for RNGs.

PractRand functions by incrementally generating bits of data with the RNGs and sending them through a variety of tests, it starts by testing 256 megabytes of data, and keeps increasing the quantity by a factor of 2 until it reaches 32 terabytes. So, with more data to find patters on, the tests will progressively get harder to pass (Doty-Humphrey, n.d.).

For this study each RNG will be put through PractRand's normal test set, this set consists of four tests, BCFN, DC6, Gap16, Brank, and FPF. Descriptions of each of these can be found on the Appendix A, however, almost all of these tests test for correlations inside an RNG's output stream in different ways and with different scopes, with tests such as the BCFN checking for long-range linear correlations, DC6 checking for short correlations.

Python's PNRG

Python has different modules for the purpose of generating random numbers, the one to be studied in this extended essay is the "random" module, since it is the one built into the standard Python library, and the documentation publicly shares its algorithm. However, Python is very clear in warning that for purposes

which require cryptographically strong random numbers the random module is not to be used, and instead users should look into the "secrets" module, for which there is no public information available.

The RNG behind python's random module is the famous "Mersenne Twister 19937" (Python, n.d.).

Mersenne Twister 19937

The Mersenne Twister 19937 or "MT19937" is a PRNG that was developed by Makoto Matsumoto and Takuji Nishimura in 1997.

The MT19937 RNG uses a deterministic algorithm to generate a sequence of random numbers. The algorithm that it uses to do so begins by generating a sequence of Mersenne numbers, these are numbers in the form $2^p - 1$, where p is a prime number. These numbers are then combined and transformed using a series of operations, including bitwise operations (operations involving the manipulation of individual bits), matrix operations, and more. The result of these operations is a sequence of numbers that appears to be random, though as with any PRNG it is actually determined by the initial seed value.

The sequence of numbers generated by the MT19937 RNG has a period of $2^{19937} - 1$, which means that the sequence will not repeat until it has generated $2^{19937} - 1$ numbers. This is where the RNG gets its name from. The RNG also uses a state of 19968 bits, represented by 624 32-bit integers. The state of an RNG is the set of internal variables which will determine the next number it will output; these variables are updated each time the RNG outputs a number. It is worth noting that the state of the RNG is directly correlated with the period, and in most cases the number of bits will be the exponent of the number 2 for the period, however there is a discrepancy of 31 bits in this case, as the MT19937 discards some of the bits to achieve a better distribution of random numbers (Wikimedia Foundation, 2002).

The MT19937 RNG is known for its high-quality random numbers, which pass a wide range of statistical tests for randomness and are generated relatively quickly. Nowadays the MT19937 is one of the most widely used RNGs, and does a great job for most practical purposes, however, there are many algorithms

which create better quality randomness, so it is no longer used for purposes which require cryptographically strong random numbers.

Testing Python's RNG

Chi-squared test

For the Chi-squared test, the RNG will generate one billion random integers from 0 to 9, and the previously explained Chi-squared test will be performed with the RNG's output.

The following script will be used for this purpose.

Chi-square test.py

```python
# Import the random module
import random

# Import the chi2 function from scipy.stats
from scipy.stats import chi2

# Initialize variables for storing chi-square values and test results
chi_square_values = []
counter = 0
failed_tests = 0
passed_tests = []

# Number of integers to be generated to one billion
sample_size = 10000000000

# Initialize a list to store the generated random integers
random_ints = []

# Generate the random integers and store them in the list
for j in range(sample_size):
    random_ints.append(random.randint(0, 9))
    counter += 1

    # Calculate and print the progress of generating random integers
    progress = (counter * 100) / sample_size
    print(f"{progress}%")

# Initialize a list to store the frequency of each integer between 0 and 9
frequencies = [0] * 10

# Calculate the frequency of each integer
for k in random_ints:
    frequencies[k] += 1

# Initialize a variable to store the chi-square value
chi_square = 0

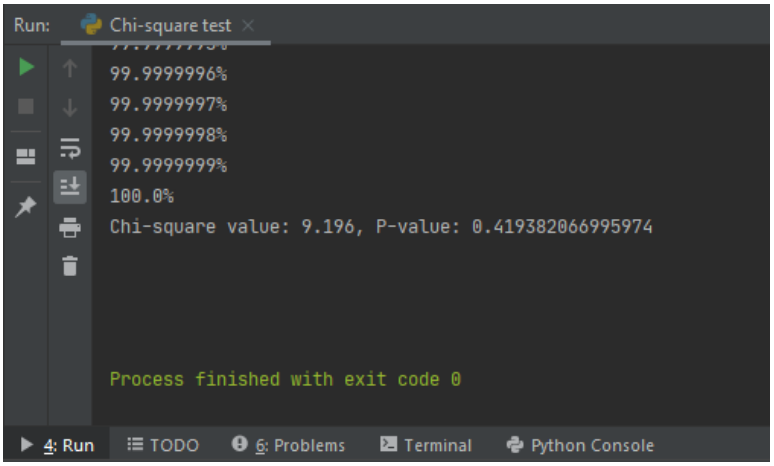
# Calculate the chi-square value
for l in frequencies:
    chi_square += (l - (sample_size / 10)) ** 2 / (sample_size / 10)

# Calculate the p-value using the chi-square value and 9 degrees of freedom
p_value = 1 - chi2.cdf(chi_square, 9)

# Print the results
print(f"Chi-square value: {chi_square}, P-value: {p_value}")

```

The output for this code is the following:



The chi-square statistic is $\simeq$ 9.20. By comparing this statistic with the critical value, one will find that $9.20 < 16.92$, meaning that the MT19937 passes the Chi-square goodness of fit test at a 0.05 significance level by a great margin.

PractRand

Running the MT19937 through PractRand is as simple as building the PractRand library and running a command. As previously explained PractRand incrementally tests the RNG by testing more data each time. Because of this the test results for the first series of tests will output results like the following:

```
RNG_test using PractRand version 0.94
RNG = mt19937, seed = 0xc0198c90
test set = core, folding = standard (32 bit)

rng=mt19937, seed=0xc0198c90
length= 256 megabytes (2^28 bytes), time= 2.1 seconds
  no anomalies in 165 test result(s)
```

Because of this, only the results where the test finds anomalies, or straight up failures will be shown. In the case of the MT19937 this occurs at 256 gigabytes.

```
rng=mt19937, seed=0x90477c69
length= 256 gigabytes (2^38 bytes), time= 2138 seconds
  Test Name                         Raw       Processed     Evaluation
  [Low8/32]BRank(12):12K(1)         R= +6116  p~=  4e-1842    FAIL !!!!!!!!
  ...and 283 test result(s) without anomalies
```

The MT19937 fails the PractRand standard testing battery at 256 gigabytes by a resounding failure on a binary rank matrix test.

JavaScript's PRNG

Determining what RNG does JavaScript use is not as simple as with Python, as the specific algorithm used is not specified in the JavaScript language specification, meaning that different JavaScript implementations may use different algorithms. As of 2022 the V8 JavaScript Engine is the engine which powers every chromium-based browser such as Google Chrome, Microsoft Edge, Brave or Opera. This engine was updated in 2015 so that the PRNG behind JavaScript's function Math.random() was Xorshift128+. On top of this other popular browsers such as Mozilla and Safari also use Xorshift128+ as their RNG, so it would make sense to take Xorshift128+ as JavaScript's RNG for this study, as it is the most widely used RNG, despite this not being specified within the language. (Williams, 2015)

Xorshift128+

The Xorshift128+ is a Xorshift or shift-register generator, a sort of PRNG invented by George Marsaglia in 2003.

Xorshift generators, contrary to the MT19937 which uses a variety of operations, only use two bitwise operations, specifically the bitwise exclusive or (XOR) operation, and the bitwise shift operation to generate a sequence of random numbers. It is from these operations that the generator class derives its name. (Marsaglia, 2003)

The XOR operation (represented as "^") compares each bit of one number with the corresponding bits of another, if two bits have the same value the resulting bit is 0, and if two bits have different values the result bit is 1, the XOR operation systematically does this for every bit of the two numbers, effectively creating a new number. For example, the 8-bit binary representation of the number "9" is "0000 1001", and the representation of the number "7" is "0000 0111", so the XOR operation would be the following:

$$9 \text{ ^ } 7 = \begin{array}{r} 0000\ 1001 \\ \text{^ } 0000\ 0111 \\ \hline 0000\ 1110 \end{array} = 14$$

(Bitwise XOR, n.d.)

The shift operation moves the bits of a number a specified number of spaces in a specified direction, discarding the bits that are shifted off the number in that direction, and adding as many 0 bits to the right as spaces the number was shifted, it is represented as "<<" or ">>" depending on the direction of the shift. For example, if the number "9" was to be shifted to the left by 2 bits, the operation would be the following:

$$9 << 2 = 0000\ 1001 << 2 = 0010\ 0100 = 36$$

(IBM, n.d.)

What distinguishes the Xorshift128+ from other members of the Xorshift family is that the algorithm's period is $2^{128} - 1$ which is directly determined by the state, represented by 128 bits in the form of two 64-bit integers.

The main appeal of the Xorshift128+ lies in its speed and convenience, with its standard implementation being twelve lines of C code, and not relying in any external libraries or modules, this algorithm is one of the fastest at the expense of some quality in its random numbers, as it has a comparatively small state (Vigna, 2017).

Testing JavaScript's RNG

Chi-squared test

The procedure for this test is the same as with python, the chi-square statistic will be obtained by performing a chi-square test with one billion randomly generated integers.

The script used for this purpose is the following:

JS Chi-squared test.js

```js
const chiSquareDist = require('chi-squared');

// Initialize variables for storing chi-square values and test results
let chiSquareValues = [];
let counter = 0;
let failedTests = 0;
let passedTests = [];

// Number of integers to be generated to one billion
const sampleSize = 1000000000;

// Initialize an array to store the generated random integers
let randomInts = [];

// Generate the random integers and store them in the array
for (let j = 0; j < sampleSize; j++) {
    randomInts.push(Math.floor(Math.random() * 10));
    counter++;

    // Calculate and log the progress of generating random integers
    let progress = (counter * 100) / sampleSize;
    console.log(`${progress}%`);
}

// Initialize an array to store the frequency of each integer between 0 and 9
let frequencies = Array(10).fill(0);

// Calculate the frequency of each integer
for (let k = 0; k < randomInts.length; k++) {
    frequencies[randomInts[k]]++;
}

// Initialize a variable to store the chi-square value
let chiSquare = 0;

// Calculate the chi-square value
for (let l = 0; l < frequencies.length; l++) {
    chiSquare += Math.pow(frequencies[l] - (sampleSize / 10), 2) / (sampleSize / 10);
}

// Calculate the p-value using the chi-square value and 9 degrees of freedom
let pValue = 1 - chiSquareDist.cdf(chiSquare, 9);

// Log the results
console.log(`Chi-square value: ${chiSquare} P-value: ${pValue}`);

```

The output for this code is the following:

```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE
99.9999994%
99.9999995%
99.9999996%
99.9999997%
99.9999998%
99.9999999%
100%
Chi-square value: 9.998 P-value: 0.3506478461215141
>
```

The resulting Chi-square statistic is $\simeq 10$. As $10 < 16.92$, the Xorshift128+ passes the Chi-square test at a significance level of 0.05, by another notable margin.

PractRand

The procedure to follow in order to run the Xorshift128+ through PractRand is slightly more complex than for the MT19937, as contrary to it, Xorshift128+ is not built into PractRand's RNG library. Because of this, in order to run it the test, the RNG's output must be piped into PractRand, which is why the RNG is set to be "RNG_stdin64" instead of "Xorshift128+". The first and final failure is the following.

```
RNG = RNG_stdin64, seed = 0x9b2f9a09
test set = normal, folding = standard (64 bit)

rng=RNG_stdin64, seed=0x86dd9a09
length= 256 megabytes (2^28 bytes), time= 2.8 seconds
  Test Name                         Raw       Processed     Evaluation
  [Low4/64]BRank(12):768(1)         R= +1272  p~=  5.4e-384   FAIL !!!!!!!
  [Low1/64]BRank(12):256(2)         R= +3748  p~=  3e-1129    FAIL !!!!!!!!
  [Low1/64]BRank(12):384(1)         R= +5405  p~=  3e-1628    FAIL !!!!!!!!
  [Low1/64]BRank(12):512(1)         R= +8161  p~=  1e-2457    FAIL !!!!!!!!
  ...and 155 test result(s) without anomalies
```

Xorshift128+ fails the PractRand standard testing battery at 256 megabytes by four resounding failures, all on binary rank matrix tests.

Comparison

Chi-squared test

As previously discussed, this chi-squared test tests the uniformity of the distribution of outcomes these RNGs output. After generating one billion random numbers with each RNG and testing the distribution of each, the results show how at a significance level of 0.05 both RNGs pass the test. As both RNGs pass the test, despite the MT19937's chi-square statistic being fractionally smaller than Xorshift128+'s, there will be no further value judgement placed upon these results as the difference is small and results could go either way in reruns of the experiment. Additionally, since both tests return chi-squared values bellow the critical

value, the p-values are considerably above the significance level, providing further evidence that the outputs are distributed equally.

Despite each RNG getting similar results in each test, it is worth noting how long did each programming language take to generate the billion integers and perform the tests. The MT19937 took about three and a half hours to generate the billion integers and perform the test, while the Xorshift128+ took roughly one. Since the focus of this experiment was not timing these processes, these are only rough estimates, and there could be many uncontrolled variables affecting the performance of each RNG such as the implementation script, or other activity in the PC which may have hindered these times. However, this showcases the main strength of the Xorshift128+, which lies in its speed.

PractRand

As previously discussed, PractRand tests for the independence of the outcomes of each RNG. After putting each RNG through the PractRand standard test battery there is a clear winner. The Xorshift128+ algorithm does not pass this battery at its smallest level of 256 megabytes, while the MT19937 reaches a whooping 256 gigabytes before failing. Interestingly, each RNG fails on Brank or binary rank matrix tests, these tests place bits on binary matrices to identify patterns which may have gone unnoticed by more traditional tests, which focus on more obvious patterns (See Appendix A).

Either way, this test shows that the output of the MT19937 is way stronger cryptographically than the Xorshift128+'s, in this case about 1024 times.

Conclusion

"Best" RNG

Based on the results of the aforementioned tests it is easy to proclaim a RNG the winner. Just by looking at the numbers it is clear that the MT19937 obtains better results than the Xorshift128+ by a long shot, however, answering the question of which algorithm is "better" requires more nuance than just looking at

the numbers. Is the MT19937 better than the Xorshift128+? Cryptographically speaking yes, it isn't close. However, the Xorshift128+ never attempts to compete in that category, its main appeal lies in its speed, as its standard implementation is only twelve lines of C code, while a typical MT19937 implementation can be several hundred lines of code (although admittedly this is not best measure of a program's speed). This feature can be seen in something as simple as how long it took for each RNG to generate the billion integers.

Additionally, the only type of test the Xorshift128+ fails at 256 megabytes are Brank matrix tests, and while this means that the RNG is extremely weak cryptographically speaking, as previously discussed these tests detects really obscure patters, which would never be detected by the human eye. This means that the algorithm could produce at least 256 megabytes of seemingly random numbers before showing obvious patterns, while keeping an equal distribution as is known thanks to the chi-squared test. To put things into perspective, 256 megabytes can store about 64,000,000 32-bit integers, this is a big number of "random numbers" to be accessed at incredibly high speeds which could be used for many non-cryptographical purposes.

On the other hand, the MT19937 provides a good balance between cryptographical robustness and relative speed, with a very uniform outcome distribution and 256 gigabytes before failing a binary rank matrix test (the equivalent to about 64,000,000,000 32-bit integers by the same measure as before).

Finally, context matters, the main use the Xorshift128+ receives is in JavaScript, a language used to "create dynamic and interactive web content like applications and browsers" (Meltzer, 2020), because of this its fast nature makes it the best fit for the job. Meanwhile, the MT19937 is used widely for many different intents and purposes, its role is being a jack of all trades, which it fulfills perfectly for its aforementioned qualities.

Personal verdict

When it comes to answering my own question of which RNG does the job best, this investigation has revealed that the MT19937 does. However, this does not mean that I will exclusively use this RNG going

forward. I believe that each algorithm is smartly put in place to play its key role. When I'm coding a webpage I don't really care about the quality of the numbers as they are not used for anything important, I just need a big volume of random numbers and I want them to load fast. On the other hand, the range of projects I might tackle with python is wider, which is why I think it is good to have a multipurpose RNG which will ensure quality while being fast.

Overall, this investigation has been a success as it has helped me understand how random numbers are obtained in deterministic machines, as well as why each RNG is put in its place. Looking back there are some improvements to be made, mainly I would have liked to precisely test for the speed of each RNG, as it ended up being such a key factor, however without having performed the tests I never anticipated this would be what separates an RNG from the other.

Thanks to this investigation, going forward I will not exclusively use one RNG, and I will be mindful of why I am deciding to use one over the other, making sure to use the correct tool for each job.

Appendix A: Practrand Standard Test Battery

Test:	Description:
BCFN	The BCFN, or "Binary code frequency of nybbles" test checks for long range linear correlations in the RNG's data stream. A nybble is a group of four binary digits, also known as a "half-byte", what BCFN does is examine the uniformity of the distribution of gaps between occurrences of the same nybble in a sequence of random numbers. A uniform distribution of gaps is an indicator of high-quality randomness while a non-uniform one is the opposite. This particular test is effective against Lagged Fibonacci generators (LFG), this is a type of PRNG which generates numbers based on the Fibonacci sequence, but contains a lag, this lag indicates how many of the previous numbers are going to be added to generate the next one. LFGs with big lags can usually bypass other statistical tests.
DC6	The DC6, or "Diffusion of the 6^{th} order cumulative sums" test checks for short range independence in the RNG's data stream. Independence refers to the degree to which the values of one random variable are unaffected by other random variables. The test is based on the concept of cumulative sum, a sequence of numbers calculated by adding up the values in a given sample. This test calculates the cumulative sum for a sample of binary data and then evaluates the quality of randomness by comparing it to the expected distribution for a random sample, a degree of deviation from the expected distribution is calculated and used to evaluate the RNG. This test is similar to a chi-square test as it checks for an uneven distribution, however, it is also useful for detecting specific patters or biases, while being pretty efficient as it only requires calculating the cumulative sum for the sample instead of checking each value in the sample.
Gap16	A form of gap test, this test checks for patterns or regularities in the distribution of the generated numbers. To perform the gap test, PractRand divides the range of possible numbers to be generated into a number of intervals, and then counts the number of generated numbers that fall into each interval. The test then checks for any gaps or patterns in the distribution of numbers across the intervals. If the distribution is uniform and random, there should be no gaps or patterns.
Brank	A binary matrix rank test. This test consists of generating a large number of random bits using the RNG, and then arranging these bits into a binary matrix (a rectangular array of elements in which these elements are either 0 or 1). The test calculates the rank of the matrix, which is a measure of the number of independent rows or columns in the matrix. This can be done in a variety of ways, but the simplest one is by reducing the matrix to echelon form. In order for a matrix to be in echelon for it needs to meet two conditions. 1. All of the non-zero rows are above any rows of all zeros (if there are any). 2. The leading entry (first non-zero number) of any non-zero row is in a column to the right of the leading entry of the row above it. In this way an example of a binary matrix in echelon form could be the following:

	$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$ Once one puts the matrix in echelon form using a variety of equations, the rank of the matrix can be determined by counting the number of non-zero rows, in the example above, the rank of the matrix would be 2. A high-quality random number generator should produce a matrix with a high rank, which indicates that the rows and columns of the matrix are independent and contain no discernible patterns or regularities. The test is useful because it can detect certain types of non-randomness that may not be detectable by other tests.
FPF	The FPF, or "floating point frequency" test checks for extremely short-term correlations (shorter than the DC6 test) in the data stream. The test counts the number of floating-point numbers within a given range, and then compares the observed frequency to the expected frequency based on the distribution of the numbers, again, in a similar fashion to a chi-squared test. The test is applied to the binary format of the floating-point numbers, which means that the test is based on the binary representation of the numbers, rather than on the decimal representation. The test uses overlapping windows of the original data stream. This means that the test is applied to overlapping segments of the data stream, rather than to the entire stream as a whole. This allows the test to evaluate the randomness and quality of the numbers in different parts of the data stream, and to identify potential patterns or biases that may exist in the generated numbers.

(Doty-Humphrey, n.d.)

Bibliography

Bitwise XOR (^) - javascript: MDN. JavaScript | MDN. (n.d.). Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_XOR

Dolenc, S. (2022, July 8). *What is randomness?* Sci-highs. Retrieved from https://sci-highs.com/what-is-randomness/

Doty-Humphrey, C. (n.d.). *PractRand*. Pracrand.sourceforge.net. Retrieved from https://pracrand.sourceforge.net/PractRand.txt

Doty-Humphrey, C. (n.d.). *PractRand Tests Overview*. Pracrand.sourceforge.net. Retrieved from https://pracrand.sourceforge.net/Tests_overview.txt

Haahr, M. (n.d.). *True random number service*. RANDOM.ORG - True Random Number Service. Retrieved from https://www.random.org/

IBM. (n.d.). *Bitwise left and right shift operators << >>*. IBM Documentation. Retrieved from https://www.ibm.com/docs/en/i/7.3?topic=expressions-bitwise-left-right-shift-operators

JMP. (n.d.). *Chi-square goodness of fit test*. Statistics knowledge portal. Retrieved from https://www.jmp.com/en_sg/statistics-knowledge-portal/chi-square-test/chi-square-goodness-of-fit-test.html

Johnston, N. (n.d.). *Chi-square distribution calculator*. Statdistributions. Retrieved from https://www.statdistributions.com/chisquare/

Khan Academy. (n.d.). *Pseudorandom number generators | Computer Science | Khan Academy*. *Khan Academy*. Retrieved from https://www.khanacademy.org/computing/computer-science/cryptography/crypt/v/random-vs-pseudorandom-number-generators.

Marsaglia, G. (2003). Xorshift rngs. *Journal of Statistical Software, 8*(14). https://doi.org/10.18637/jss.v008.i14

Mindrila, D., & Balentyne, P. (n.d.). *The Chi Square Test* . Retrieved from https://aep.westga.edu/academics/research/vrc/assets/docs/ChiSquareTest_LectureNotes.pdf

Meltzer, R. (2020, December 3). *What is JavaScript used for?* Lighthouse Labs. Retrieved from https://www.lighthouselabs.ca/en/blog/what-is-javascript-used-for

Merriam-Webster. (n.d.). *Computer Definition & meaning*. Merriam-Webster. Retrieved from https://www.merriam-webster.com/dictionary/computer

Python. (n.d.). *Random - generate pseudo-random numbers*. Python documentation. Retrieved from https://docs.python.org/3/library/random.html

Science direct. (n.d.). *Radioactive decay*. Science direct. Retrieved from https://www.sciencedirect.com/topics/mathematics/radioactive-decay

Sidhpurwala, H. (2019, June 5). *Understanding random number generators, and their limitations, in linux*. Red Hat - We make open source technologies for the enterprise. Retrieved from https://www.redhat.com/en/blog/understanding-random-number-generators-and-their-limitations-linux

Vigna, S. (2017). Further scramblings of Marsaglia's xorshift generators. *Journal of Computational and Applied Mathematics*, *315*, 175–181. https://doi.org/10.1016/j.cam.2016.11.006

Walker, J. (1996, May). *Hotbits: Genuine random numbers, generated by radioactive decay*. HotBits. Retrieved from https://www.fourmilab.ch/hotbits/

Weisstein, E. W. (n.d.). *Chi-squared distribution*. Wolfram MathWorld. Retrieved from https://mathworld.wolfram.com/Chi-SquaredDistribution.html

Wikimedia Foundation. (2002, March 22). *Mersenne twister*. Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Mersenne_Twister

Williams, E. (2015, December 28). *V8 Javascript fixes (horrible!) random number generator*. Hackaday. Retrieved from https://hackaday.com/2015/12/28/v8-javascript-fixes-horrible-random-number-generator/