Comparison of pseudo random number generators

1 Introduction

This report will discuss various pseudo-random number generating (PRNG) algorithms, comparing their performance and accuracy. Each algorithm will have a brief introduction, followed by an implementation of the algorithm in C. We will then test each algorithm with the very popular KS (Kolmogorov–Smirnov) test and also attempt to do a visual analysis. We will also compare these algorithms on their performance. More information on the tests will be provided below. After our analysis, we will discuss the algorithms used by the standard random function of various languages and also some of the general applications of PRNGs.

1.1 What are pseudo-random number generators?

Pseudo-random number generating algorithms (or PRNGs, as they will be referred to from now on) are algorithms that given an initial number called a seed, will generate a sequence of numbers that have properties similar to a sequence of actual random numbers. It is important to note that PRNGs do not produce a truly random sequence of numbers. Hence they are sometimes also called deterministic random bit generators (DRBG). John von Neumann, an inventor of one of the first PRNGs, very aptly said, "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin."

PRNGs, however, do have various applications, especially in fields like game development and cryptography. We will discuss these applications in slightly more detail in later sections of this report. PRNGs are mainly used for their speed/performance and due to their use of seeds, which allows us to reproduce a sequence of random numbers.

1.2 PRNGs vs. HRNGs

As mentioned earlier, PRNGs are deterministic and do not always satisfy the properties of a true sequence of random numbers. So how do we generate a *true* sequence of random numbers? These are generated by specific devices called Hardware Random Number Generators (HRNG) or True Random Number Generators (TRNG). These devices generate random numbers using physical

phenomena - usually random "noise" that is generated at a microscopic level. HRNGs are a lot more cryptographically secure when compared to PRNGs and are used in the popular Internet encryption protocol, Secure Sockets Layer (SSL).

TRNGs do not always depend on a piece of hardware, however. For example, RANDOM.ORG boasts a true random number generator that uses atmospheric noise. There have even been instances of random number generators using snapshots of "Lava Lamps" to generate random numbers! In general, TRNGs require a source of external entropy to generate sequences.

TRNGs are non-deterministic and aperiodic, but they are computationally expensive. This is why they are widely used to generate a random seed for a PRNG, while the PRNG actually generates a sequence of random numbers using this seed.

1.3 Seeds

Seeds are just numbers that are used by PRNGs to generate a sequence of random numbers. The sequence produced by a seed will always be the same for a particular implementation of an algorithm.

Seeds are often obtained from the state of the computer system, like the time. For TRNGs they are obtained from a lot more random states, like atmospheric noise.

2 Testing PRNGs

The goal of this project is to test various PRNGs and compare them based on their randomness and performance. Note that this project does not attempt to do a statistical analysis of these algorithms, but rather understand the algorithms and use tests to provide evidence for their behaviour.

If you would like to test the algorithms manually, you can run `python3 test.py` *after* compiling the algorithm and running it once (you will also have to install the *numpy* and *scipy* packages for python). Compiling and running the algorithm will write the random numbers to a file called 'rand.txt', which the test file will automatically use to run the tests. You can also generate a visual bitmap of the numbers by running `python3 vis.py` (again, remember to install the required packages). This will create a JPEG image in the same folder which will contain the bitmap produced by that algorithm. An algorithm can be compiled by using the command `gcc -Wall -Werror -O -o algo_name algo_name.c` and run using the command `./algo_name`.

Note that all the seeds have been hardcoded, so you should get the same results while testing that can be seen in this report.

2.1 Testing randomness - KS Test

For testing randomness, we will use a frequency test called the KS test or the Kolmogorov-Smirnov test. This test is named after Andrey Kolmogorov and Nikolai Smirnov. It is one of the most useful tests for comparing two samples; it quantifies the distance between the empirical distribution function of the sample and cumulative distribution function of the reference sample.
(The equations below are from Wikipedia)

The empirical distribution function F_n for n ordered observations X_i is defined as

$$F_n = \frac{1}{n} \sum_{i=1}^{n} I_{[-\infty,x]}(X_i)$$

where $I_{[-\infty,x]}(X_i)$ is the indicator function, equal to 1 if $X_i \leq x$ and equal to 0 otherwise.

The KS statistic for a given cumulative distribution function $F(x)$ is

$$D_n = \sup_x |F_n(x) - F(x)|$$

where $\sup_x$ is the supremum of the set of distances.

For this project, we will be using the Python package *scipy* to carry out the KS test. The test will be two-sided and we will compare our empirical values with same number of values from a uniform distribution (and with the same range).

2.2 Visual testing

We will also attempt to visually test the randomness of the algorithms by creating a bitmap of the random numbers generated by the algorithm. This is done by scaling the random numbers down to [0,1] and picking black or white for a pixel depending on if the scaled number is greater than 0.5 or not.

2.3 Testing performance

We will be testing the performance of each algorithm by checking how much time it takes to generate a sequence of 1,000,000 random numbers.

3 Analysis

We will be analysing the following algorithms:

- Middle Square Method (1946 - John von Neumann)
- Lehmer/Park-Miller Generator (1951 - D.H. Lehmer/1988 - S.K. Park, K.W. Miller)
- Linear Congruential Generator (1958 - W. E. Thomson, A. Rotenberg)

- Linear-feedback shift register (1965 - R.C. Tausworthe)
- Mersenne Twister (1998 - M. Matsumoto, T. Nishimura)
- Xorshift (2003 - G. Marsaglia)
- Xoroshiro128+ (2018 - D. Blackman, S. Vigna)

3.1 Middle Square Method

The middle square method is one of the first known PRNGs. It is not a very good generator and has a low period, which results in the sequence repeating itself.

The middle square method uses a number with n digits, which is squared to create a number with $2n$ digits. If the square is not $2n$ digits long, it is padded with 0s. The middle n digits would be the next number in the sequence, and the process is repeated to generate new numbers. However, this method may not converge to 0 and also has issues with limited periodicity if n is odd. Hence, we will be working with a slightly modified version called the Middle Square Weyl Sequence PRNG, which allows convergence to 0 and prevents the program from running indefinitely. The "middle" in this method is extracted by shifting the number right by 32 bits.

We will be using a C implementation as below to test the algorithm. The seed here is $0xb5ad4eceda1ce2a9$, which is the default. (Source: Wikipedia)

```c
#include <stdint.h>

uint64_t x = 0, w = 0, s = 0xb5ad4eceda1ce2a9;

inline static uint32_t msws() {

   x *= x;
   x += (w += s);
   return x = (x>>32) | (x<<32);

}
```

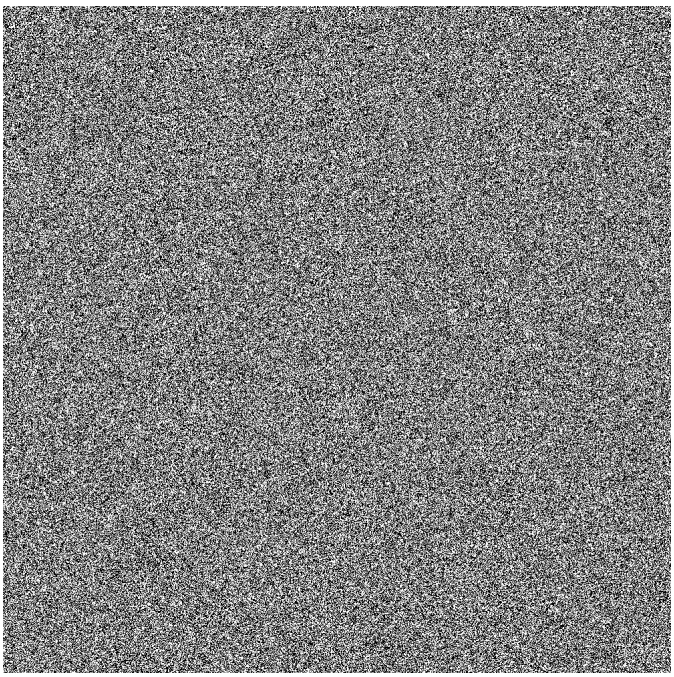
We ran the algorithm using *gcc* 5.4.0. It took an average of **0.135** seconds to generate a million random numbers. The results of the KS test and various other stats can be found below.

n	1000000
p-value	0.089
ks-statistic	0.0018
min	-2147481735
max	2147468688
mean	-1942899.948517
variance	1.5358401270182705e+18

As we can see, the p-value is around 0.089, which is greater than the required value of 0.05 that we use to reject the null hypothesis. Thus, we cannot reject

the null hypothesis, and we conclude the Middle Square method has produced a fairly uniform set of random numbers, although the p-value is still quite low compared to other generators.

Bitmap:



3.2 Lehmer/Park-Miller Generator

The Lehmer generator is a special case of a Liner congruential generator, which we will analyse next. It is one of the most influential PRNGs, and named after Derrick Henry "Dick" Lehmer. It is also known as the Park-Miller random number generator after Stephen K. Park and Keith W. Miller after they modified an initial version, and this new version is now built-in the C and C++ languages as the function *minstd*.

The general formula for the Lehmer Generator is as follows:

$$X_{k+1} = g \cdot X_k \bmod n$$

where the modulus n is a prime number or a power of a prime number, the multiplier g us an element of high multiplicative order modulo n and the seed X_0 is coprime to n.

Miller and Park's initial paper can be found here.

A C implementation is rather simple (Source: Wikipedia):

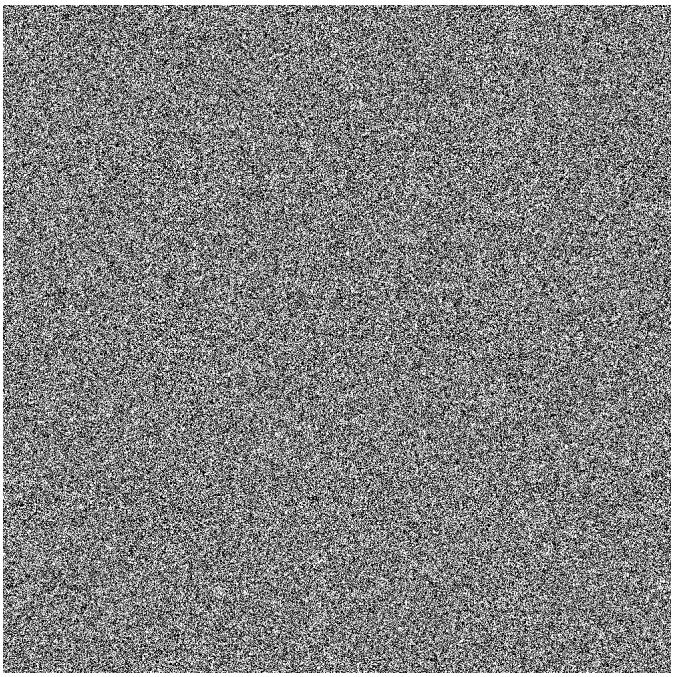
```
uint32_t lcg_parkmiller(uint32_t *state)
{
    return *state = ((uint64_t)*state * 48271u) % 0x7fffffff;
}
```

We initiate with a seed (or state) of 1. Again, we ran the algorithm using *gcc* 5.4.0 and it took an average of **0.0204** seconds to generate 1,000,000 numbers. The results of the KS test are as below.

n	1000000
p-value	0.2722
ks-statistic	0.0014
min	376
max	2147483426
mean	1073234009.472725
variance	3.839834599635939e+17

The p-value is much higher than the middle square method and ks-statistic is lower as well, which makes the Lehmer generator comfortably a better PRNG than the middle square PRNG. The higher p-value indicates that the number generated are more uniformly distributed when compared to the middle square algorithm.

Bitmap:



3.3 Linear Congruential Generator (LCG)

The LCG is a very similar algorithm to the Lehmer Generator, in fact, the Lehmer Generator is just a special case of the LCG, as we will see below:

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

where X is the sequence of pseudo-random values, m is the modulus, a is the multiplier, c is the increment and X_0 is the seed.

We will implement and test a very common LCG used by the java.util.Random class and also POSIX, with $m = 2^{48}$, $n = 25214903917$ and $c = 11$. Once again, our C implementation is not very difficult (derived from the Lehmer generator's implementation):

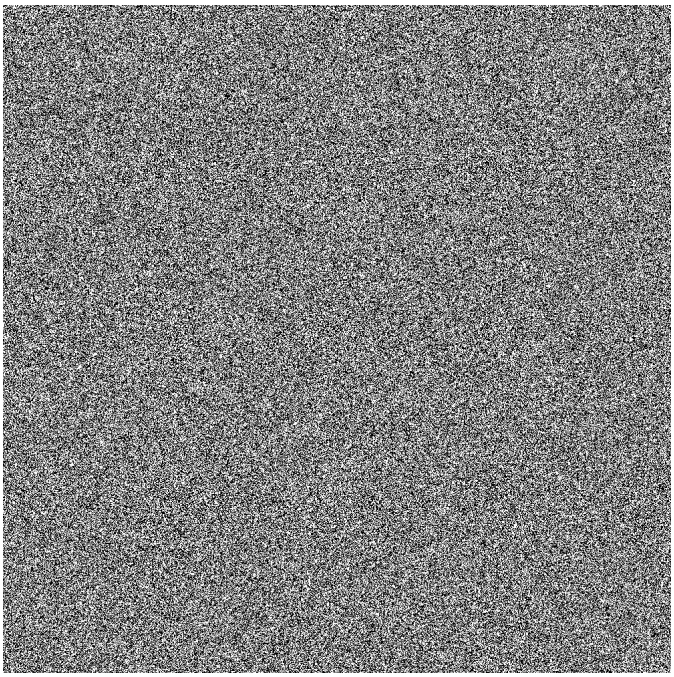
```
uint64_t lcg(uint64_t *state)
{
    return *state = ((uint64_t)*state * 25214903917 + 11) %
    281474976710656;
}
```

As earlier, we used a starting state or seed of 1. Using the same version of *gcc* as above, the algorithm took an average of **0.1625** seconds to generate a million numbers. The test results can be seen in the table below.

n	1000000
p-value	0.897
ks-statistic	0.0008
min	488198547
max	281474907015427
mean	140756095098958.38
variance	6.604056054866791e+27

The generic LCG has vastly improved upon the results we saw in the Lehmer Generator with a much higher p-value and a lower ks-statistic. We are unable to reject the null hypothesis in this case as well.

Bitmap:



3.4 Linear-feedback shift register (LFSR)

LSFRs works in a similar fashion to LCGs, in the sense that the output is a linear function of the previous state.

As the name suggests, the linear function involves a lot of shifting of bits; a lot of common LSFRs also use exclusive-or (XOR) as a linear function. LSFRs have a huge number of applications - from cryptography to circuit-testing. The CRC checks used in network protocols are also closely related to LSFRs.

We will attempt to implement and test a simple Fibonacci variant of an LSFR. In an LSFR, the bit positions that affect the next state are called taps. The arrangement of these taps can be expressed as a polynomial mod 2. This basically means that the coefficients of the polynomial will be 1 or 0. As an example, if the taps are at the 16^{th}, 14^{th}, 13^{th} and 11^{th} bits, the "feedback" polynomial is:

$$x^{16} + x^{14} + x^{13} + x^{11} + 1$$

Tausworth's paper can be found here.

We will use the taps [22, 21, 0] while implementing our very own LSFR, as these are the taps that will give us 2 million random numbers, which is closest to what we want (1 million). (Source for taps: Wikipedia)

```
1 # include <stdint.h>
2 # include <stdio.h>
3
4 int main(void)
5 {
```

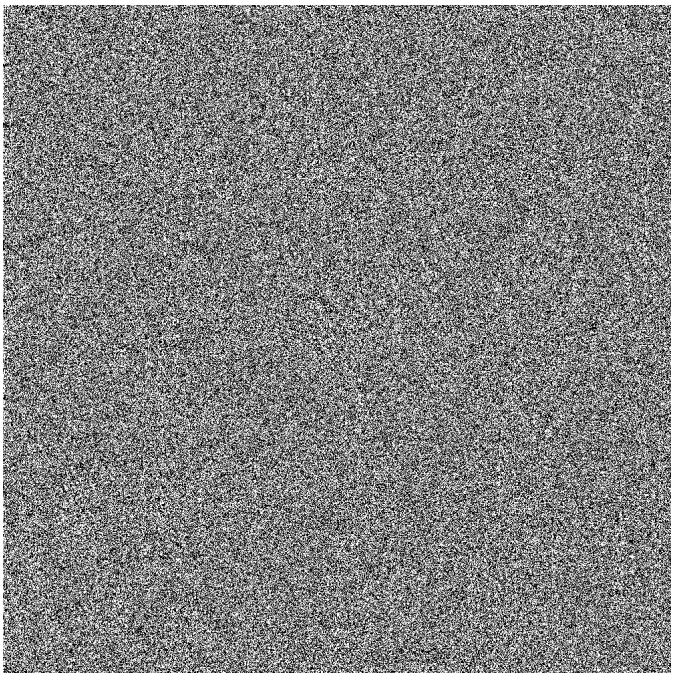
```
  // 2^22 is start_state
  uint64_t start_state = 0x007FFFFF;  /* Any nonzero start state
    will work. */
  uint64_t lfsr = start_state;
  uint64_t bit;
  unsigned period = 0;

  do
  {
    /* taps: 22 21 feedback polynomial: x^22 + x^21 + 1 */
    bit  = ((lfsr >> 0) ^ (lfsr >> 1)) & 1;
    lfsr =  (lfsr >> 1) | (bit << 22);
    ++period;
  } while (lfsr != start_state);

  printf("%d\n", period);
  return 0;
}
```

Our version of the LFSR took, on average, 0.035 seconds to generate around 2 million numbers. We can safely assume it to take around **0.0175** seconds to generate a million numbers. Further test results can be seen below:

n	2088705
p-value	0.9951
ks-statistic	0.0004
min	1
max	8388607
mean	4194261.3301514573
variance	5864159464063.818

Again, we cannot reject the null hypothesis due to a high p-value. Although LFSRs are deterministic, they are uniformly distributed.

Bitmap:



3.5 Mersenne Twister

The Mersenne Twister is probably the most famous PRNG. It is widely used as a general purpose PRNG in various languages and tools. The Mersenne Twister was developed by Makoto Matsumoto and Takuji Nishimura in 1997 to rectify flaws in previous PRNGs.

The idea is based upon previous LFSRs, except that the output of the recurrence relation is twisted using an invertible matrix called a tempering matrix. The most famous Mersenne Twister is based on the Mersenne prime $2^{19937} - 1$.

The initial paper for the Mersenne Twister can be found here.

A C implementation is as below (An implementation with detailed comments can be found in the files attached to this report).

```c
#include <stdio.h>
#include <stdlib.h>

typedef unsigned long uint32;

#define N               (624)
#define M               (397)
#define K               (0x9908B0DFU)
#define hiBit(u)        ((u) & 0x80000000U)
#define loBit(u)        ((u) & 0x00000001U)
#define loBits(u)       ((u) & 0x7FFFFFFFU)
#define mixBits(u, v)   (hiBit(u)|loBits(v))

static uint32   state[N+1];
```

```
15 static uint32   *next;
16 static int      left = -1;
17
18 void seedMT(uint32 seed)
19  {
20     register uint32 x = (seed | 1U) & 0xFFFFFFFFU, *s = state;
21     register int    j;
22
23     for(left=0, *s++=x, j=N; --j;
24         *s++ = (x*=69069U) & 0xFFFFFFFFU);
25  }
26
27
28 uint32 reloadMT(void)
29  {
30     register uint32 *p0=state, *p2=state+2, *pM=state+M, s0, s1;
31     register int    j;
32
33     if(left < -1)
34         seedMT(4357U);
35
36     left=N-1, next=state+1;
37
38     for(s0=state[0], s1=state[1], j=N-M+1; --j; s0=s1, s1=*p2++)
39         *p0++ = *pM++ ^ (mixBits(s0, s1) >> 1) ^ (loBit(s1) ? K : 0
    U);
40
41     for(pM=state, j=M; --j; s0=s1, s1=*p2++)
42         *p0++ = *pM++ ^ (mixBits(s0, s1) >> 1) ^ (loBit(s1) ? K : 0
    U);
43
44     s1=state[0], *p0 = *pM ^ (mixBits(s0, s1) >> 1) ^ (loBit(s1) ?
    K : 0U);
45     s1 ^= (s1 >> 11);
46     s1 ^= (s1 <<  7) & 0x9D2C5680U;
47     s1 ^= (s1 << 15) & 0xEFC60000U;
48     return(s1 ^ (s1 >> 18));
49  }
50
51
52 uint32 randomMT(void)
53  {
54     uint32 y;
55
56     if(--left < 0)
57         return(reloadMT());
58
59     y  = *next++;
60     y ^= (y >> 11);
61     y ^= (y <<  7) & 0x9D2C5680U;
62     y ^= (y << 15) & 0xEFC60000U;
63     return(y ^ (y >> 18));
64  }
65
66 int main(void)
67  {
68     FILE *f = fopen("rand.txt", "w+");
```

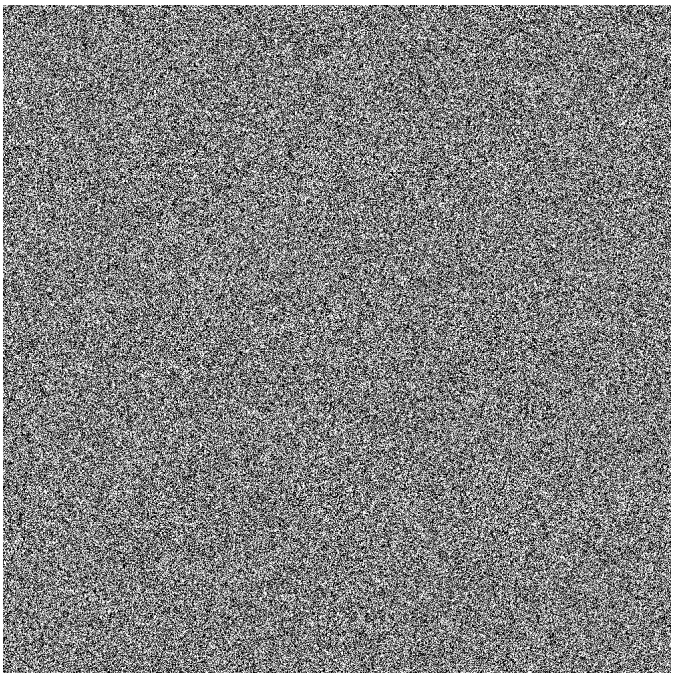
```
69      if (f == NULL) {
70          printf("Error opening file!\n");
71          return 0;
72      }
73      int j;
74
75      // you can seed with any uint32, but the best are odds in
        0..(2^32 - 1)
76
77      seedMT(42);
78
79
80      for(j=0; j<1000000; j++) {
81          randomMT();
82      }
83
84      return(EXIT_SUCCESS);
85  }
```

The Mersenne Twister took an average of **0.0199** seconds to generate a million random numbers. The KS test results are represented in the table below.

n	1000000
p-value	0.9562
ks-statistic	0.0007
min	13614
max	4294967216
mean	2148220316.251655
variance	1.5378170629960732e+18

Once again, we observe a high p-value and a low KS-Statistic. We are able to reject the null hypothesis again.

Bitmap:



3.6 XorShift

Xorshift is not a particular algorithm, rather a class of algorithms based on LFSRs we studied earlier. This class of algorithms were first discovered by George Marsaglia, his paper can be found here.

Very similar to the LFSRs, they're known for their high performance, albeit they are not cryptographically safe. They use a similar bit shifting method and the parameters have to be chose carefully to achieve a long period.

We will attempt to test a slight modification of a normal xorshift generator, a xorshift128+ generator that uses addition as a faster non-linear transformation. It is used in the JavaScript engines of Chrome, Firefox and Safari. A C implementation of xorshift128+ is as below (sourced from Wikipedia).

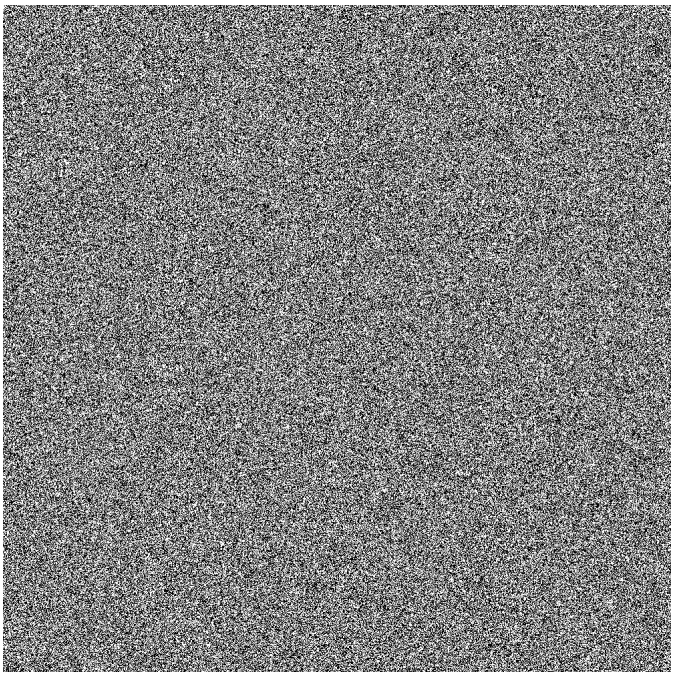
```c
#include <stdint.h>

/* The state must be seeded so that it is not all zero */
uint64_t s[2];

uint64_t xorshift128plus(void) {
  uint64_t x = s[0];
  uint64_t const y = s[1];
  s[0] = y;
  x ^= x << 23; // a
  s[1] = x ^ y ^ (x >> 17) ^ (y >> 26); // b, c
  return s[1] + y;
}
```

Running on *gcc* 5.4.0, xorshift128+ took about **0.0171** seconds to generate a million random numbers. KS test results below:

n	1000000
p-value	0.92817
ks-statistic	0.00077
min	8388677
max	1.8446722723846777e+19
mean	9.22396613549192e+18
variance	2.8365995125388314e+37

Yet again, we cannot reject the null hypothesis, and with such a high p-value we can claim that xorshift128+ produces uniformly distributed values.

Bitmap:



3.7 Xoroshiro128+

Xoroshiro128+ is one of the latest PRNGs to be developed. It was developed as a successor to xorshift128+ but it uses a shift/rotate-based linear transformation.

Working off of George Marsaglia's design, it was improved by Sebastiano Vigna and David Blackman (paper can be found here). We can see an improvement in both speed and statistical quality as well as a high period of 2^{128}.

A C implementation is as below. An implementation with more comments can be found in the files attached with this report. (Source)

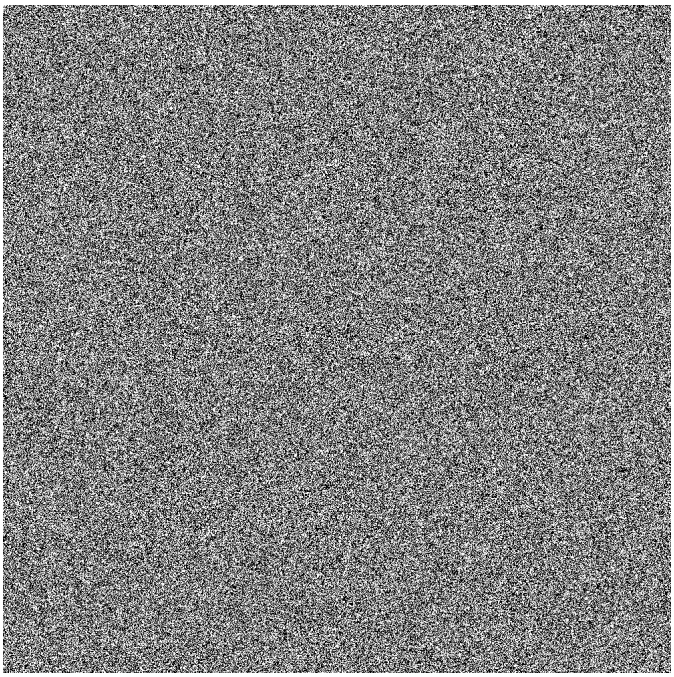
```c
#include <stdint.h>

static inline uint64_t rotl(const uint64_t x, int k) {
  return (x << k) | (x >> (64 - k));
}


static uint64_t s[2];

uint64_t next(void) {
  const uint64_t s0 = s[0];
  uint64_t s1 = s[1];
  const uint64_t result = s0 + s1;

  s1 ^= s0;
  s[0] = rotl(s0, 24) ^ s1 ^ (s1 << 16); // a, b
  s[1] = rotl(s1, 37); // c

  return result;
}

void jump(void) {
  static const uint64_t JUMP[] = { 0xdf900294d8f554a5, 0
    x170865df4b3201fc };

  uint64_t s0 = 0;
  uint64_t s1 = 0;
  for(int i = 0; i < sizeof JUMP / sizeof *JUMP; i++)
    for(int b = 0; b < 64; b++) {
      if (JUMP[i] & UINT64_C(1) << b) {
        s0 ^= s[0];
        s1 ^= s[1];
      }
      next();
    }

  s[0] = s0;
  s[1] = s1;
}

void long_jump(void) {
  static const uint64_t LONG_JUMP[] = { 0xd2a98b26625eee7b, 0
    xdddf9b1090aa7ac1 };

  uint64_t s0 = 0;
  uint64_t s1 = 0;
  for(int i = 0; i < sizeof LONG_JUMP / sizeof *LONG_JUMP; i++)
    for(int b = 0; b < 64; b++) {
      if (LONG_JUMP[i] & UINT64_C(1) << b) {
        s0 ^= s[0];
        s1 ^= s[1];
      }
      next();
    }
  s[0] = s0;
  s[1] = s1;
}
```

The algorithm took an average of **0.015** seconds to generate a million numbers, making it our fastest generator yet. The KS Test results can be observed in the table below.

n	1000000
p-value	0.9493
ks-statistic	0.00073
min	3
max	1.8446736887362157e+19
mean	9.22467874529209e+18
variance	2.8398260184838615e+37

The above table makes it clear that Xoroshiro128+ is not only extremely fast, but also very statistically sound, with a high p-value and a low KS-Statistic.

Bitmap:



4 Final Comparison

Algorithm	Time, n=1000000	p-value	KS-Statistic
Middle Square	0.135s	0.089	0.0018
Lehmer/Park Miller	0.0204s	0.2722	0.0014
LCG	0.1625s	0.897	0.0008
LFSR	0.0175s	0.9951	0.0004
MT	0.0199s	0.9562	0.0007
Xorshift128+	0.0171s	0.92817	0.00077
Xoroshiro128+	0.015s	0.9493	0.00073

Xoroshiro128+ comes out as the fastest PRNG, while surprisingly the LFSR does better on the KS-test than some of its more recent counterparts. This can be boiled down to the random sampling that is used to test against a uniform distribution, and it is clear that the other algorithms are not low performing by any means. In general, if we were to carry out a larger variety and number of tests, for example a Chi-Test and the BigCrush test suite, we would see the newer algorithms outperform the older ones.

5 PRNGs used by major languages

Language	PRNG
C	LFSR/LCG (gcc)
C++	LCG/MT (based on compiler)
Java	LCG/Park-Miller
Python	MT
JavaScript	xorshift128+ (Chrome, Mozilla and Safari)
Go	LFSR
MATLAB	MT
Mathematica	LCG/MT
PHP	MT
R	MT (among various others)
Ruby	MT
Rust	xorshift128+

6 Applications of RNGs

Randomness has a variety of applications across various fields. Since time immemorial, humankind has used randomness to its benefit. The Ancient Greeks were one of the firsts, using random sampling to elect governments.

But as time has passed, uses of randomness have become more and more sophisticated, especially when it comes to science. As it has become clear, statistic sampling is one of the most important uses of randomness. Along with sampling, simulations have also become very reliant on good random generators.

The Monte Carlo methods are a very good example of this; these algorithms try to use repeated random sampling to solve deterministic problems.

With so many cryptocurrencies popping up these days, it is also important to mention cryptography as an application of randomness. Although PRNGs aren't cryptographically safe due to their deterministic nature, randomness is required to generate public & private keys and nonces.

Another popular application is in games. A lot of AIs in games, especially earlier ones, used randomness to determine their behaviour. These days we have more sophisticated uses, such as procedural generation. Procedural generation is the method of creating game assets algorithmically rather than manually. *Minecraft* is a very popular game that uses procedural generation and *No Man's Sky* is another game in which you can explore 18 quintillion procedurally generated planets.

7 Conclusion

In this report, we analysed and compared various PRNGs across the ages, from Neumann's middle square method to the brand new xoroshiro128+. We used the KS test to check if the results were uniformly distributed and produced bitmaps to visually compare outputs. We were able to draw some interesting conclusions from our results, with most newer algorithms doing very well on the KS-Test. After the analysis, we discussed the PRNGs used in programming applications and the various applications of randomness.