Uniwersytet Wrocławski

Pseudorandom Number Generators

Authors: Łukasz Stodółka, Łukasz Besuch, Marcin Rypuła

Wrocław, 2026

Contents

1 What is a Pseudorandom Number Generator?

Pseudorandom number generators (PRNGs) play a fundamental role in computer science, numerical methods, simulations, cryptography, randomized algorithms, and statistical experiments [1, 2]. Their purpose is to generate sequences of numbers which, although produced by a deterministic algorithm, imitate certain properties of genuinely random sequences. In contrast to true randomness, which is usually associated with physical phenomena and inherently unpredictable processes, pseudorandomness is purely algorithmic. A PRNG is completely determined by its initial configuration. Therefore, once the initial value, called the seed, is fixed, the entire generated sequence can be reproduced exactly. This reproducibility is one of the main reasons why PRNGs are widely used in simulations, debugging, benchmarking, and experimental research. The central idea behind pseudorandom number generation is not to produce randomness in a philosophical or physical sense, but rather to produce deterministic sequences whose statistical behavior is sufficiently close to that of random sequences for a given application.

1.1 Formal Definition

Formally, a PRNG can be described as a discrete dynamical system consisting of:

- a finite state space S,
- a deterministic transition function $f : S \to S$,
- an output function $g : S \to \mathbb{R}$.

The generator is defined by the recurrence relation:

$$x_{n+1} = f(x_n),$$

with initial state (seed) $x_0 \in S$, and output sequence:

$$u_n = g(x_n).$$

Here, (x_n) represents the internal state sequence, while (u_n) represents the generated pseudorandom numbers.

1.2 Seed (Initial Value)

The seed x_0 is the initial element of the state space S. It uniquely determines the entire sequence generated by the PRNG.

If the same seed is used multiple times, the generator produces the same sequence, which is a key property used for reproducibility in simulations, debugging, and numerical experiments.

Different seeds lead to different trajectories in the state space, increasing variability of outputs.

1.3 State Space and Periodicity

Since the state space S is finite and the generator is deterministic, the sequence (x_n) must eventually repeat. This follows from the pigeonhole principle: after a finite number of steps, some state must reoccur, causing the sequence to enter a cycle.

Thus, every PRNG becomes eventually periodic.

The period of a PRNG is the smallest positive integer T such that:

$$x_{n+T} = x_n \quad \text{for all } n \geq n_0,$$

for some transient length $n_0 \geq 0$.

A good PRNG is characterized by a very long period, making repetition practically unobservable in applications.

1.4 Uniformity and Equidistribution

Uniformity refers to how evenly the output values are distributed over a given interval.

For normalized outputs $u_n \in [0, 1]$, a sequence is considered uniformly distributed if, for any subinterval $[a, b] \subset [0, 1]$, we have:

$$\lim_{N\to\infty} \frac{1}{N} \#\{1 \leq n \leq N : u_n \in [a, b]\} = b - a.$$

This property is also known as *equidistribution*.

Lack of uniformity introduces bias and can significantly distort results in simulations and statistical applications, such as Monte Carlo methods.

1.5 Computational Randomness

Although PRNGs are deterministic, their goal is to produce sequences that are statistically indistinguishable from truly random sequences under practical tests.

In other words, while they are not random in a physical sense, they exhibit *computational randomness*, meaning that without knowledge of the seed, the sequence appears random for algorithmic and statistical purposes.

1.6 Motivation for Implementation

A pseudorandom number generator can therefore be understood as a deterministic algorithm defined on a finite state space, designed to produce sequences with desirable

statistical properties. A good general-purpose PRNG should satisfy several important requirements:

- deterministic behavior, allowing reproducibility,
- a sufficiently long period,
- efficient computation,
- good uniformity and equidistribution properties,
- absence of easily detectable patterns,
- suitability for the intended application.

Many classical pseudorandom number generators can be described within this general framework. Examples include linear congruential generators, Lehmer generators, xorshift generators, and LFSR-based generators. They differ mainly in the choice of the transition function f, the structure of the state space S, and the output function g.

In the following sections, selected pseudorandom number generators will be implemented and analyzed in C++. Classical surveys and testing frameworks emphasize exactly this combination of theory, implementation and empirical validation [3, 2]. Before introducing particular algorithms, we first define a common interface that all generators will satisfy. This interface provides a uniform way to initialize generators, produce values, collect statistics, and generate visualizations.

1.7 General Generator Interface

In order to compare different pseudorandom number generators in a consistent way, all implementations in this project follow a common interface. This interface abstracts away the concrete internal mechanism of a generator and focuses only on the operations required to use and analyze it.

Conceptually, each generator is treated as an object with an internal state. The state can be initialized using a seed, advanced to produce the next output value, and described by basic information about its output range.

In the C++ implementation, this idea is expressed using a generic `RandomGenerator` concept. A type satisfying this concept must provide the following operations:

- `next()`, which advances the generator and returns the next pseudorandom value,
- `seed(seed)`, which initializes or resets the internal state,
- `name()`, which returns the name of the generator,

- `min()` and `max()`, which describe the range of possible output values,
- `outputBits()`, which specifies how many bits of output are produced by a single call to `next()`.

This interface is directly connected to the mathematical model introduced earlier. The method `seed()` selects the initial state x_0, while `next()` corresponds to applying the transition function f and then producing an observable output using the output function g.

Therefore, although different generators may use different recurrence relations and internal representations, they can still be tested and analyzed using the same code. This makes it possible to compare their periods, distributions, bit-level behavior, histograms, and correlations under a unified experimental framework.

Such an approach is also useful from a software engineering perspective. It separates the implementation of a particular generator from the code responsible for statistical analysis and visualization. As a result, adding a new PRNG requires only implementing the required interface, while the existing analysis tools can be reused without modification.

Listing 1: Plik `codes/RandomConcept.cpp`

```
using SEED = uint64_t;

template <typename G>
concept RandomGenerator =
requires(G gen, SEED seed) {
    { gen.next() } -> std::convertible_to<uint64_t>;
    { gen.seed(seed) } -> std::same_as<void>;
    { gen.name() } -> std::convertible_to<std::string_view>;
    { gen.min() } -> std::convertible_to<uint64_t>;
    { gen.max() } -> std::convertible_to<uint64_t>;
    { gen.outputBits() } -> std::convertible_to<int>;
};
```

2 Classical Methods of Pseudorandom Number Generation

Classical pseudorandom number generators are based on simple mathematical constructions, typically involving modular arithmetic and linear recurrence relations. Despite their simplicity, they form the basis for many modern designs and provide important insight into the structure of deterministic random-like sequences.

In this section, we present three fundamental classes of classical generators: the Linear Congruential Generator (LCG), the Lehmer generator, and combined generators.

2.1 Linear Congruential Generator (LCG)

The Linear Congruential Generator is one of the simplest and most widely studied pseudorandom number generators [3, 1]. It is defined by the recurrence relation:

$$x_{n+1} = (ax_n + c) \bmod m,$$

where:

- x_n is the current state,
- a is the multiplier,
- c is the increment,
- m is the modulus,
- x_0 is the seed.

The state space of an LCG is the finite ring $\mathbb{Z}_m$, and thus all values satisfy $x_n \in \{0, 1, \ldots, m-1\}$.

2.1.1 Period of LCG

The maximum possible period of an LCG is m. This full period is achieved if and only if the following conditions hold:

- $\gcd(c, m) = 1$,
- $a - 1$ is divisible by all prime factors of m,
- if m is divisible by 4, then $a - 1$ is also divisible by 4.

These conditions are known as the Hull–Dobell theorem, which characterizes full-period LCGs [3].

2.1.2 Properties and Limitations

LCGs are computationally efficient and easy to implement. However, their statistical quality depends strongly on the choice of parameters.

A well-known drawback is that sequences generated by LCGs exhibit lattice structure when considered in higher dimensions. This geometric regularity reduces their suitability for high-dimensional simulations such as Monte Carlo methods.

Additionally, the period may be significantly smaller than m if parameters are poorly chosen.

2.1.3 Example of LCG

Example with parameters $m = 9$, $a = 2$, $c = 1$, $x_0 = 1$:

n	x_n
0	1
1	3
2	7
3	6
4	4
5	0
6	1

Table 1: Example sequence generated by an LCG

2.1.4 C++ Implementation

The theoretical recurrence relation of the Linear Congruential Generator can be translated directly into C++. The internal state of the generator is stored as an unsigned integer, while the parameters a, c, and m are represented respectively by the multiplier, increment, and modulus.

The method `next()` applies the recurrence

$$x_{n+1} = (ax_n + c) \bmod m$$

and returns the newly computed state.

Listing 2: Core `next()` function of the Linear Congruential Generator

```
uint64_t next() {
    __uint128_t value = static_cast<__uint128_t>(_multiplier) * _state +
    _increment;
    _state = static_cast<uint64_t>(value % _modulus);
    return _state;
}
```

The implementation is a direct translation of the mathematical formula. First, the current state is multiplied by the multiplier a, then the increment c is added, and finally the result is reduced modulo m. The new value becomes the next internal state and is returned as the output of the generator. The use of `__uint128_t` prevents overflow during the intermediate multiplication before the modulo operation is applied.

2.1.5 Experimental Comparison

To illustrate the influence of parameters, three LCG variants were compared. The first generator uses parameters satisfying the Hull–Dobell conditions. The second generator uses a large modulus but poorly chosen parameters. The third generator differs from the good one only by one parameter: the increment is even, so the condition $\gcd(c, m) = 1$ is violated.

Parameter	Good LCG	Bad large-range LCG	One broken condition
Seed	123456789	123456789	123456789
Multiplier a	1103515245	65539	1103515245
Increment c	12345	0	12344
Modulus m	2^{31}	2^{31}	2^{31}

Table 2: LCG parameter sets used in the experiment

For each generator, 100000 generated values were used to compute basic statistics. The histogram was divided into 20 buckets. The sequence plot and the successive-value plot were generated using 5000 values.

Statistic	Good LCG	Bad large-range LCG	One broken condition
Min	14984	26007	18045
Max	2147472790	2147458605	2147483589
Mean	1072058253.99	1073741022.56	1072768794.75
Std. dev.	618784777.74	619155200.59	618797107.84
One-bit ratio	0.499968	0.532329	0.499702
χ^2	17.2004	11.7680	14.9208

Table 3: Basic statistics computed from 100000 LCG outputs

The good LCG has a mean close to the middle of the interval $[0, 2^{31} - 1]$ and a one-bit ratio almost equal to 0.5. This suggests that, at least in this simple experiment, zeros and ones appear in the binary representation with nearly equal frequency.

The bad large-range LCG shows why simple statistics can be misleading. Its minimum, maximum, mean and standard deviation do not look catastrophic. Even the chi-square value for the histogram is not extremely large. However, the one-bit ratio is equal to 0.532329, which is visibly shifted away from the ideal value 0.5. This indicates imbalance at the bit level.

The generator with one broken condition looks relatively close to the good one in these basic statistics. This is important, because it shows that a generator can fail a theoretical condition without immediately looking broken in a short one-dimensional test. The problem is structural: when $\gcd(c, m) \neq 1$, the full period is not guaranteed.

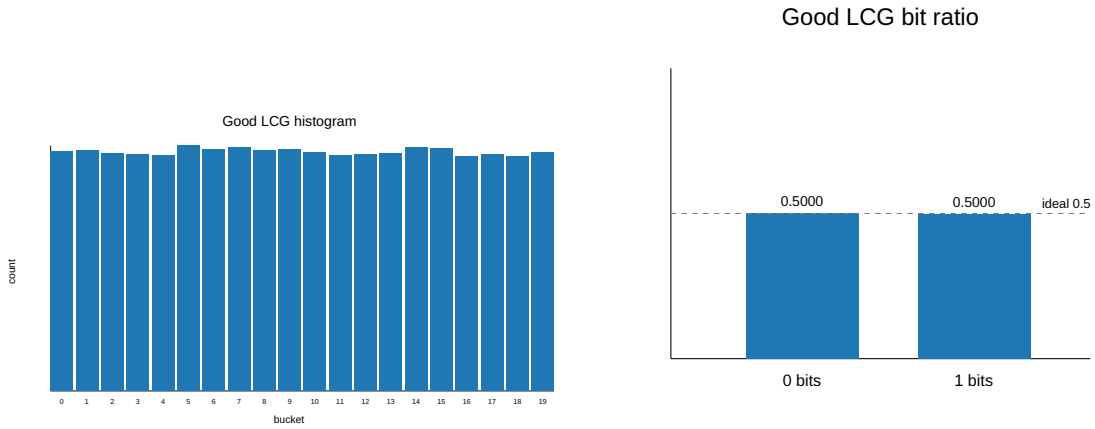

Figure 1: Histogram and bit ratio for the good LCG

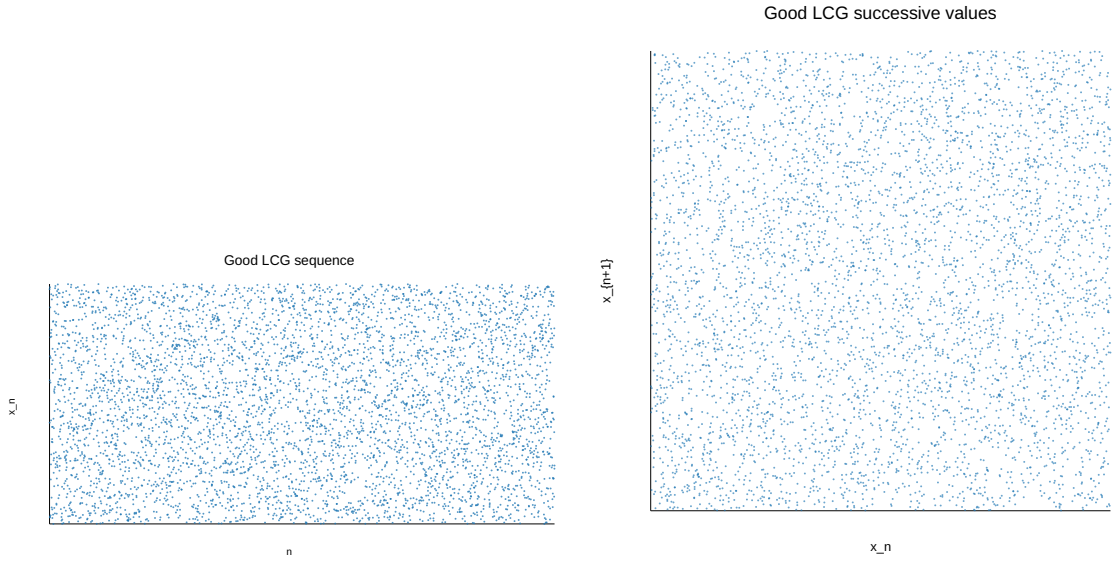

Figure 2: Sequence plot and successive-value plot for the good LCG

Figures 1 and 2 show that the good LCG covers the output range evenly in this experiment. The histogram is balanced, the bit ratio is close to 0.5, and the successive-value plot does not collapse into a small visible subset of points.

Figures 3 and 4 show that a large output range alone does not guarantee good randomness. The generator still produces values over a large interval, but the bit-ratio plot reveals a visible bias caused by the poor choice of parameters.

Figures 5 and 6 show that a single broken period condition may still produce visually acceptable plots. This confirms that empirical plots are useful for detecting obvious

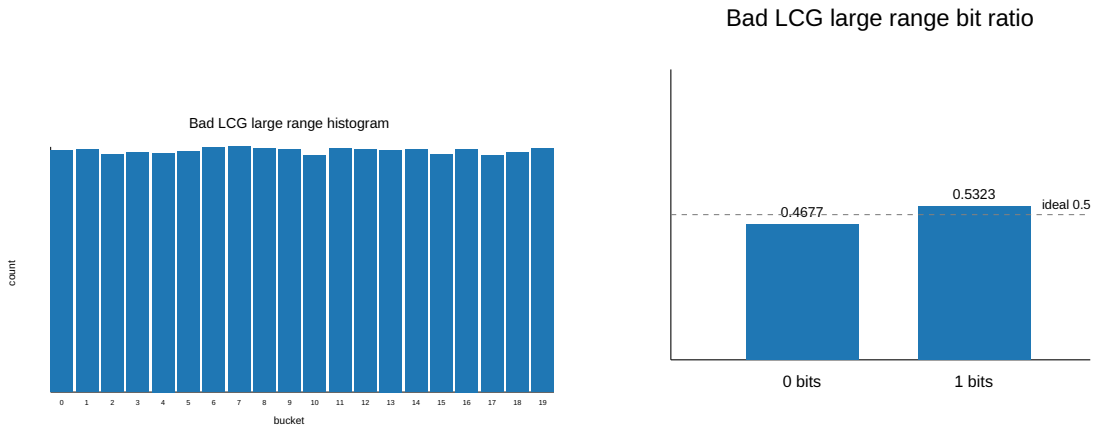

Figure 3: Histogram and bit ratio for the bad large-range LCG

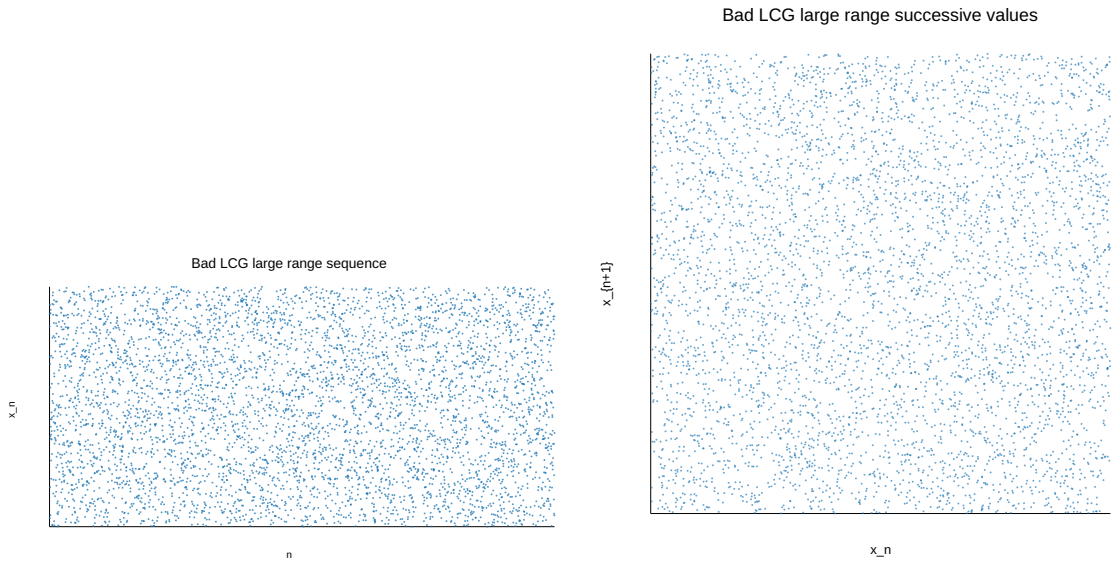

Figure 4: Sequence plot and successive-value plot for the bad large-range LCG

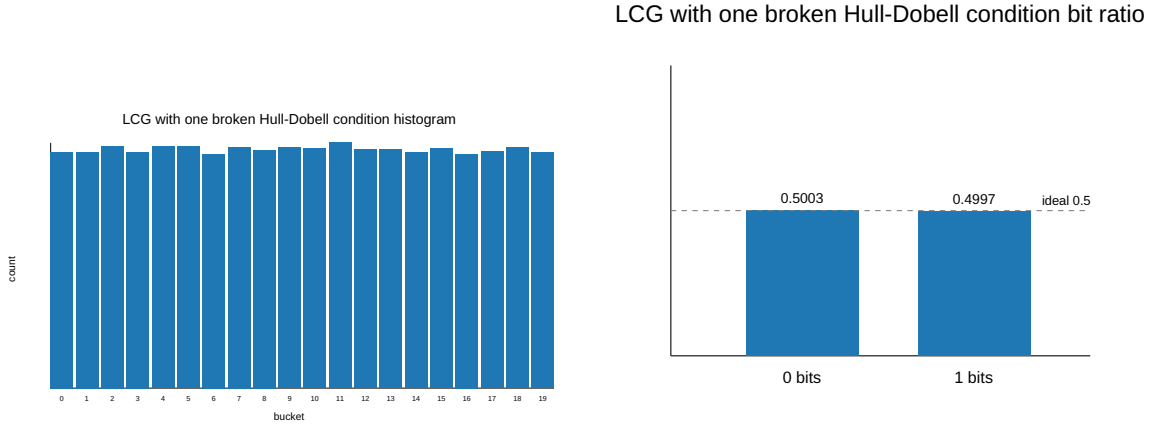

Figure 5: Histogram and bit ratio for the LCG with one broken Hull–Dobell condition

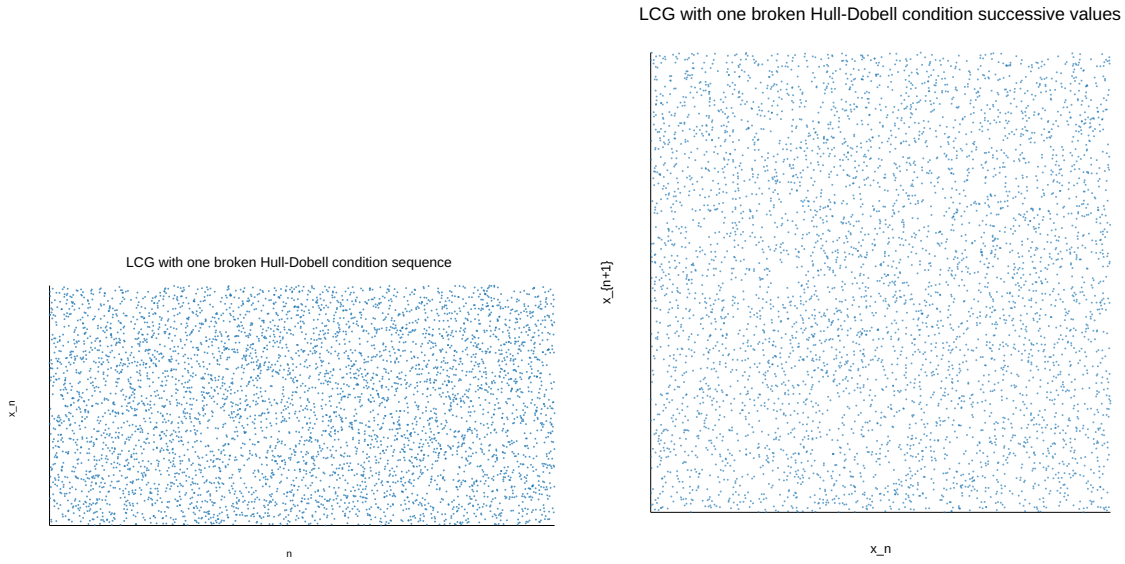

Figure 6: Sequence plot and successive-value plot for the LCG with one broken condition

problems, but they do not replace theoretical period analysis.

2.2 Lehmer Generator

The Lehmer generator is a special case of the LCG where the increment $c = 0$ [4]. It is defined by:

$$x_{n+1} = (ax_n) \bmod m.$$

It is also known as the multiplicative congruential generator.

In this case, the state evolves in the multiplicative group $\mathbb{Z}_m^*$.

2.2.1 Period

To achieve a long period, the modulus m is typically chosen as a prime number, and the multiplier a is chosen as a primitive root modulo m.

Under these conditions, the generator achieves the maximum period:

$$T = m - 1.$$

2.2.2 Properties

- simpler structure compared to full LCG,
- good statistical properties for appropriate parameter choices,
- still vulnerable to correlation effects and structural artifacts.

2.2.3 C++ Implementation

The Lehmer generator uses the same modular idea as LCG, but without an additive increment. Therefore, the transition consists only of multiplication followed by reduction modulo m.

Listing 3: Core `next()` function of the Lehmer generator

```
uint64_t next() {
    __uint128_t value = static_cast<__uint128_t>(_multiplier) * _state;
    _state = static_cast<uint64_t>(value % _modulus);
    return _state;
}
```

The implementation multiplies the current state by the multiplier a and reduces the result modulo m. As in the LCG implementation, an extended integer type is used for the intermediate product to avoid overflow before the modulo operation. Since there is no increment, the zero state is absorbing: if the state ever becomes zero, all future values are also zero. For this reason, Lehmer generators are usually analyzed on the non-zero residues modulo m.

2.2.4 Experimental Comparison

Three Lehmer variants were compared. The good variant uses the classical MINSTD-style parameters with prime modulus $m = 2147483647$ and multiplier $a = 48271$, a parameter choice in the family of multiplicative congruential generators studied in numerical simulation literature [5]. The bad large-range variant uses $a = m - 1$, which produces a very short cycle alternating between two values. The single-bad-parameter variant keeps the same large prime modulus but uses a non-primitive multiplier, which preserves the large numerical range while damaging the period structure.

Parameter	Good Lehmer	Bad large-range Lehmer	Non-primitive multiplier
Seed	123456789	123456789	123456789
Multiplier a	48271	2147483646	1073741823
Modulus m	2147483647	2147483647	2147483647

Table 4: Lehmer parameter sets used in the experiment

The good Lehmer generator has a balanced bit ratio and a low chi-square value, which suggests a relatively even one-dimensional distribution in this experiment. The bad large-range variant is intentionally deceptive: the values are still large and the bit ratio is exactly balanced, but the chi-square value is enormous. This happens because the sequence alternates between only two values, so almost all histogram buckets remain

Statistic	Good Lehmer	Bad large-range Lehmer	Non-primitive multiplier
Min	930	123456789	106782249
Max	2147479582	2024026858	2040701398
Mean	1074185909.16	1073741823.50	1073746274.59
Std. dev.	621925036.58	950285034.50	570682073.19
One-bit ratio	0.499607	0.500000	0.500000
χ^2	15.3896	900000.0000	13424.2256

Table 5: Basic statistics computed from 100000 Lehmer outputs

empty. The non-primitive multiplier variant is less trivial, but the chi-square value is still very large, indicating a serious distribution problem.

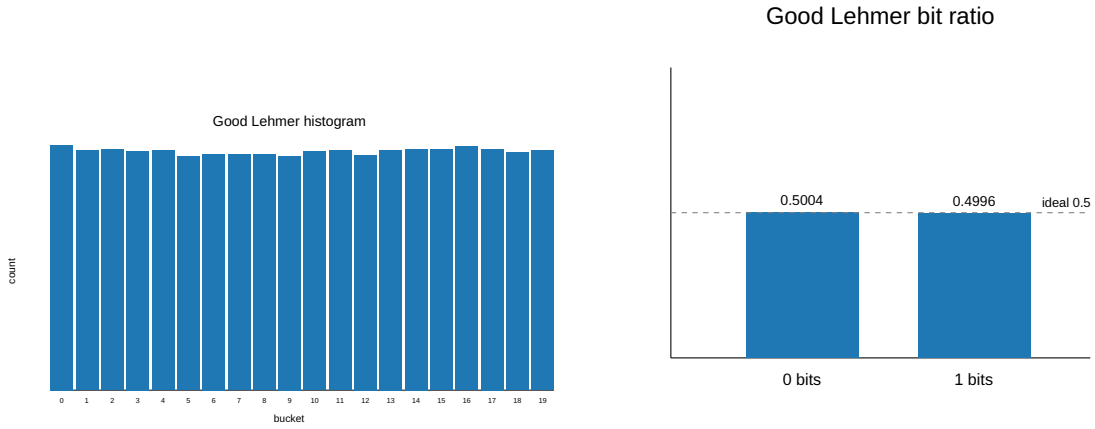

Figure 7: Histogram and bit ratio for the good Lehmer generator

Figures 7 and 8 show that the good Lehmer generator spreads values over the whole interval and keeps the bit ratio close to the ideal value 0.5.

Figures 9 and 10 show the weakness of choosing $a = m - 1$. Although the numerical range looks large, the generator has period 2 for a non-zero seed. The sequence plot alternates between two levels, and the successive-value plot collapses into a tiny set of points.

Figures 11 and 12 show a less trivial but still defective case. The generator is not stuck in a two-value cycle, but the histogram and chi-square value reveal that the distribution is far from the good variant. This demonstrates why the multiplier should be chosen carefully, preferably as a primitive root modulo the prime modulus.

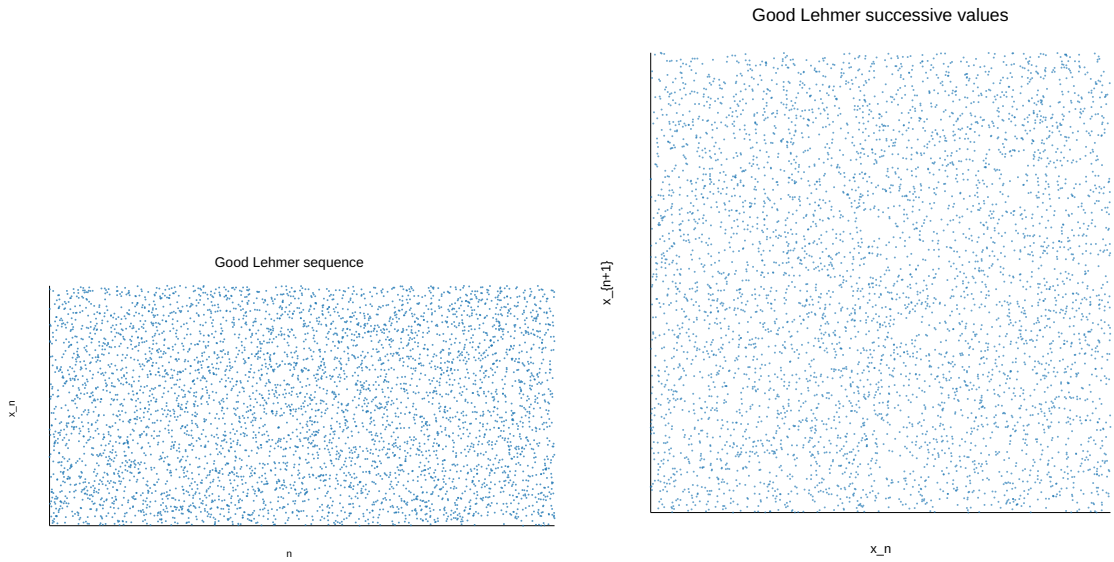

Figure 8: Sequence plot and successive-value plot for the good Lehmer generator

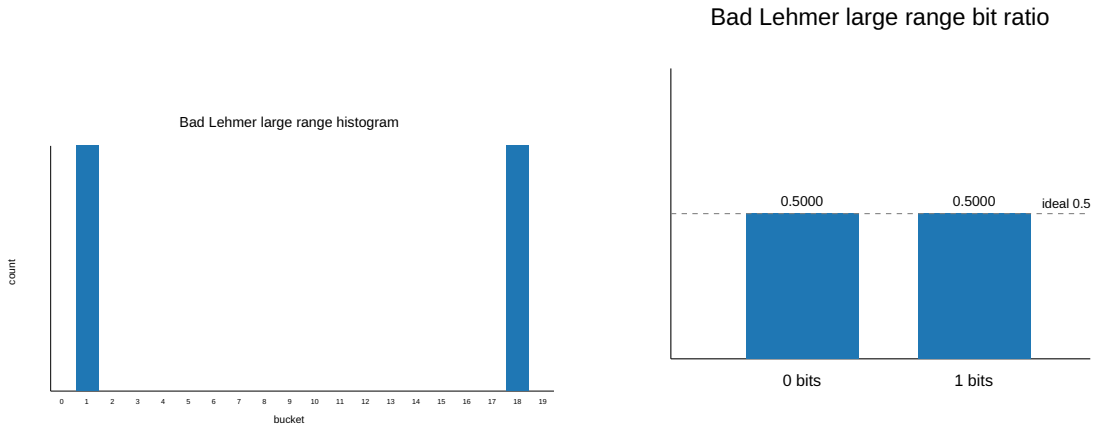

Figure 9: Histogram and bit ratio for the bad large-range Lehmer generator

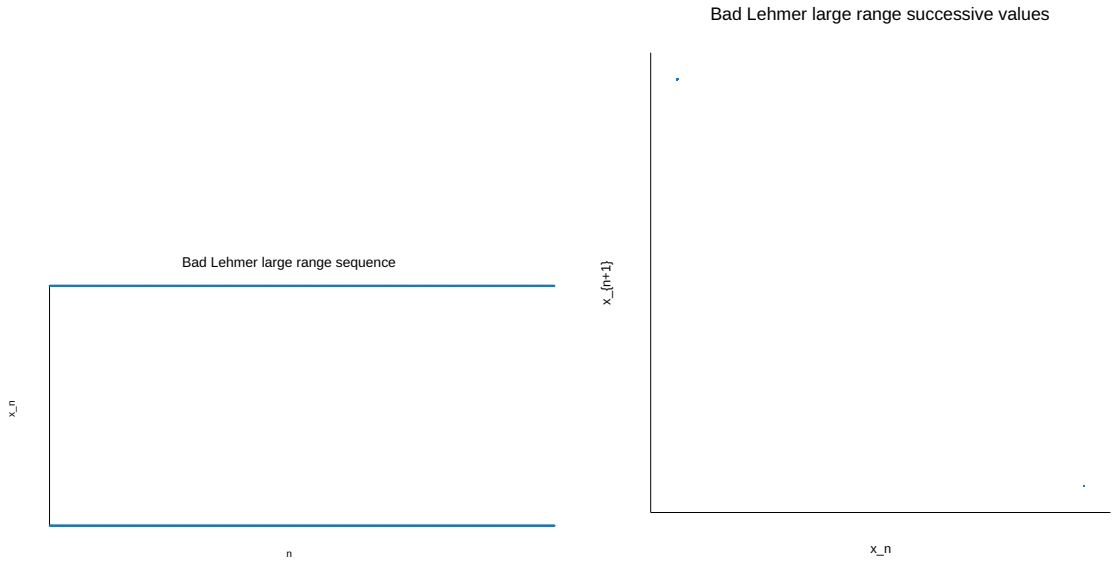

Figure 10: Sequence plot and successive-value plot for the bad large-range Lehmer generator

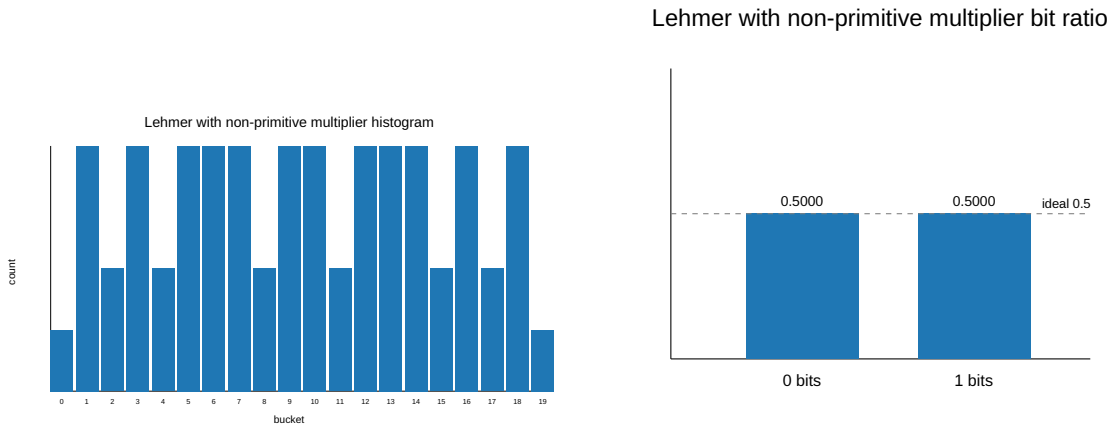

Figure 11: Histogram and bit ratio for the Lehmer generator with a non-primitive multiplier

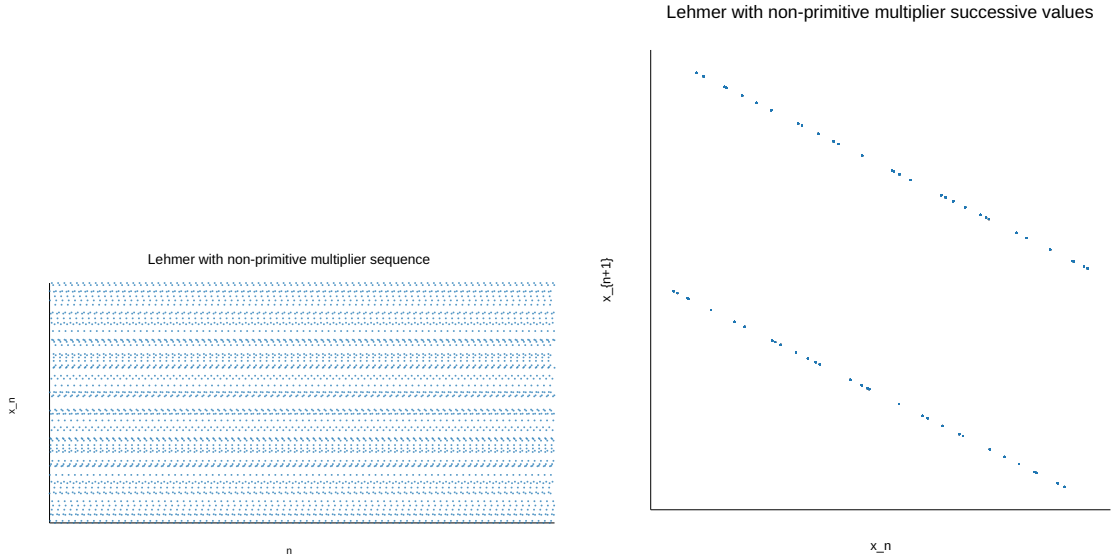

Figure 12: Sequence plot and successive-value plot for the Lehmer generator with a non-primitive multiplier

2.3 Combined Generators

Combined generators are constructed by combining multiple simpler generators in order to improve statistical properties and increase the effective period.

A typical construction is:

$$x_n = (x_n^{(1)} - x_n^{(2)}) \bmod m,$$

where $x_n^{(1)}$ and $x_n^{(2)}$ are independent sequences generated by two separate generators.

2.3.1 Period

If the individual generators have periods T_1 and T_2, then the period of the combined generator satisfies:

$$T \mid \operatorname{lcm}(T_1, T_2),$$

meaning that T divides the least common multiple of the component periods.

2.3.2 Advantages

- increased effective period,
- improved statistical behavior compared to single generators,
- reduced correlation between successive outputs.

2.4 Summary

Classical pseudorandom number generators such as LCG and Lehmer generators are simple and efficient but exhibit structural weaknesses, particularly in higher dimensions.

Combined generators partially address these issues by increasing period length and reducing correlation effects, but they still inherit linearity-based limitations.

These classical methods provide the foundation for understanding more advanced pseudorandom number generators introduced in later chapters.

3 Recursive Generators: LFSR and LFG

3.1 Linear Feedback Shift Register (LFSR)

It is incredibly fast, operates fundamentally on binary numbers (0s and 1s), and is widely used in digital electronics and simple cryptography. An LFSR consists of:

1. **Shift Register:** A device that shifts its contents into adjacent positions in the register. A bit at the end of the register is shifted out, while a new bit is inserted at the opposite end.
2. **Output Bit:** The bit at the end that is shifted out of the register.
3. **Feedback Function:** A function that takes bits from certain positions, called **taps**, combines them using the XOR operation, and feeds the result back into the register's input position. The bits are collected before the shift.

An n-bit LFSR has 2^n possible states. However, because the all-zero state will produce zeros indefinitely, the maximum possible period of an LFSR sequence is $2^n - 1$.

3.2 The Mathematics of LFSR

3.2.1 The Recurrence Relation

The new bit b_n in a Linear Feedback Shift Register (LFSR) is calculated recursively based on the previous bits. Given a register of length k, with feedback taps defined by coefficients c_i, where $c_i = 1$ if the corresponding position is an active tap and $c_i = 0$ otherwise, the sequence generation is governed by the following formula:

$$b_n = (c_1 b_{n-1} + c_2 b_{n-2} + \cdots + c_k b_{n-k}) \pmod 2.$$

Note: In binary mathematics, addition modulo 2 is equivalent to the logical XOR operation.

3.2.2 Polynomial Representation and State Shifts

This feedback relationship can be elegantly modeled using polynomials over the binary field. Let us define the characteristic polynomial p and its truncated variant $\tilde{p}$ as:

$$p(x) = x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$$

and

$$\tilde{p}(x) = c_{n-1}x^{n-1} + \cdots + c_1 x + c_0.$$

If the current state of the register is represented as a polynomial q, then:

$$q(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_0.$$

A single shift of the register corresponds to multiplying the state polynomial by the indeterminate x:

$$xq(x) = a_{n-1}x^n + a_{n-2}x^{n-1} + \cdots + a_0x.$$

To keep the state within the bounds of the register, we reduce this result modulo the characteristic polynomial p:

$$xq(x) \pmod{p(x)} = xq(x) - a_{n-1}p(x).$$

Expanding this modular reduction yields the polynomial configuration for the subsequent state:

$$\begin{aligned} xq(x) \pmod{p(x)} = {} & (a_{n-2} - a_{n-1}c_{n-1})x^{n-1} \\ & + (a_{n-3} - a_{n-1}c_{n-2})x^{n-2} + \ldots \\ & + (a_0 - a_{n-1}c_1)x - a_{n-1}c_0. \end{aligned}$$

Since the calculations are performed over $GF(2)$, subtraction is equivalent to addition modulo 2.

This polynomial operation is adjoint to the operation of the generator. More precisely, the action of the generator on the state can be written in matrix form:

$$X_n = AX_{n-1},$$

where

$$X_n = (b_n, b_{n-1}, \ldots, b_{n-k+1})^T.$$

The operation on polynomials, written in matrix form, has the transposed matrix:

$$Y_n = A^T Y_{n-1}.$$

A matrix has a maximal period if and only if its transposed matrix has a maximal period. Therefore, it is sufficient to calculate the period for the transposed matrix.

3.2.3 Algebraic Structure and Finite Fields

To analyze the periodicity and structural properties of the generated sequences, we look at the underlying algebraic frameworks. By the Chinese Remainder Theorem for integers, we know that if m and n are coprime, there exists an isomorphism:

$$\mathbb{Z}_{mn} \to \mathbb{Z}_m \times \mathbb{Z}_n,$$

defined by

$$k \to (k \bmod m,\ k \bmod n).$$

Because m and n are coprime, meaning that $\gcd(m, n) = 1$, this representation is a one-to-one mapping and, by cardinality, a bijection. An analogous property holds for the polynomial ring over a field K, provided that p_1 and p_2 are coprime:

$$K[x]/(p_1 p_2) \cong K[x]/(p_1) \times K[x]/(p_2).$$

If the characteristic polynomial p of degree n is irreducible over $K = \mathbb{Z}_2$, the quotient ring forms the finite field

$$F = K[x]/(p),$$

containing exactly 2^n elements. For any non-zero element $y \in F \setminus \{0\}$, Lagrange's theorem implies that its multiplicative order divides the order of the multiplicative group. Therefore:

$$y^{2^n-1} = 1.$$

Consequently:

$$y^{2^n-1} - 1 = 0 \implies \forall y \in F \setminus \{0\}, \quad y^{2^n} - y = 0.$$

The equality $y^{2^n} - y = 0$ also holds for $y = 0$.

3.2.4 The Role of the Least Common Multiple (LCM)

When analyzing elements in a direct product of structures, such as

$$y \in G_1 \times G_2,$$

the periodicity of the combined system is determined by the *Least Common Multiple*, denoted by lcm.

Mathematically, if an element y_1 has a period of t_1, meaning that

$$y_1^{t_1} = 1,$$

and an element y_2 has a period of t_2, meaning that

$$y_2^{t_2} = 1,$$

then the joint element

$$y = (y_1, y_2)$$

will return to the identity state only when both coordinates simultaneously complete full

cycles. This global period is the smallest positive integer L that is a multiple of both t_1 and t_2:

$$y^{\text{lcm}(t_1,t_2)} = (1,1).$$

If t_1 and t_2 share common factors, then

$$\text{lcm}(t_1,t_2) < t_1 t_2,$$

which reduces the total period of the system. The maximum sequence length is achieved when the component periods are coprime, in which case:

$$\text{lcm}(t_1,t_2) = t_1 t_2.$$

3.2.5 Periodicity and Minimal Polynomials

To determine whether an LFSR sequence can achieve its maximum length, we investigate whether there exists a smaller exponent

$$l < 2^n - 1$$

such that

$$y^l = 1$$

for all non-zero elements $y \in F$.

If

$$\forall y \in F, \quad y^{l+1} - y = 0,$$

then the polynomial

$$g(Y) = Y^{l+1} - Y$$

has all 2^n elements of F as roots. Since a non-zero polynomial of degree $l+1$ over a field can have at most $l+1$ distinct roots, it follows that:

$$l + 1 \geq 2^n.$$

Therefore:

$$l \geq 2^n - 1.$$

Because the multiplicative group of the finite field F is cyclic, there exists a primitive generator element $y \in F$ whose successive powers

$$y^k, \qquad k = 0, \ldots, 2^n - 2,$$

are all distinct and represent all non-zero elements of F.

Let h be the minimal polynomial of y. By definition, $h(y) = 0$, h is monic, and its degree is minimal among all non-zero polynomials g satisfying $g(y) = 0$.

Consequently, h must be irreducible. If there exists some polynomial g such that $g(y) = 0$, applying the polynomial division algorithm yields:

$$g = qh + r, \qquad \deg(r) < \deg(h).$$

Evaluating this equality at y gives:

$$g(y) = q(y)h(y) + r(y) \implies 0 = 0 + r(y).$$

Since $r(y) = 0$ and the degree of r is strictly less than the degree of the minimal polynomial h, it must follow that $r = 0$. Therefore, g is an exact multiple of h:

$$g = qh.$$

The evaluation mapping

$$Y \to y$$

establishes an isomorphism from the quotient ring

$$K[Y]/(h(Y))$$

to the finite field F. Most importantly, the successive powers of the residue class of Y generate every non-zero element of

$$K[Y]/(h(Y))$$

if and only if the generator has a maximum period.

Definition 3.1. *A **primitive polynomial** is an irreducible polynomial each of whose roots generates the multiplicative group of the corresponding finite field.*

Conclusion: To achieve the maximum possible period of 2^n-1 states without entering a shorter cycle, the characteristic polynomial $p(x)$ must be a **primitive polynomial** over $GF(2)$.

3.3 Example: 4-Bit LFSR

Consider a 4-bit LFSR with feedback taps at positions 3 and 4, utilizing XOR as the feedback function.

The final row repeats the initial state, showing that the LFSR returns to its starting configuration after visiting all $2^4 - 1 = 15$ non-zero states.

a	b	c (tap)	d (tap)	Output
1	0	0	1	1
1	1	0	0	0
0	1	1	0	0
1	0	1	1	1
0	1	0	1	1
1	0	1	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1
0	1	1	1	1
0	0	1	1	1
0	0	0	1	1
1	0	0	0	0
0	1	0	0	0
0	0	1	0	0
1	0	0	1	1

Table 6: Example of a 4-bit LFSR with taps c and d

3.4 LFSR C++ Implementation

The implementation follows the recurrence relation directly. In every call of `next()`, the feedback bit is computed as the XOR of all selected tap bits. Then the current register is shifted by one position, the feedback bit is inserted into the most significant position, and the result is masked so that the state remains inside the selected bit width.

Listing 4: Implementation of the LFSR transition function

```
uint64_t next() {
    uint64_t feedback = 0;

    for (int tap : _taps) {
        feedback ^= (_state >> tap) & 1ULL;
    }

    _state = (_state >> 1) | (feedback << (_width - 1));
    _state &= _mask;
    return _state;
}
```

The important part of this implementation is that the output is not produced by an external random source. It is only the result of deterministic bit operations applied to the current state. Therefore, the choice of taps and the initial state decide whether the generator has a long cycle or quickly falls into a visible pattern.

3.5 LFSR Experimental Comparison

Three LFSR variants were compared. The first one uses a 16-bit register with a tap set that gives good behaviour in this simple experiment. The second one uses a larger 32-bit state, but only one feedback tap, which creates a very short and structured cycle. The third one changes only the tap set in the 16-bit case, showing how a small structural change can strongly affect the output.

Parameter	Good 16-bit LFSR	Bad 32-bit single-tap LFSR	Single bad tap 16-bit LFSR
Seed	$0xACE1$	$0xACE1ACE1$	$0xACE1$
Width	16	32	16
Taps	$\{0, 2, 3, 5\}$	$\{0\}$	$\{0\}$
Output range	$[0, 2^{16} - 1]$	$[0, 2^{32} - 1]$	$[0, 2^{16} - 1]$

Table 7: LFSR parameter sets used in the experiment

For each generator, 100000 values were used to compute basic statistics. The histogram was divided into 20 buckets. The sequence plot and the successive-value plot were generated using 5000 values.

Statistic	Good 16-bit LFSR	Bad 32-bit single-tap LFSR	Single bad tap 16-bit LFSR
Min	1	224857447	3431
Max	65535	3786203564	57772
Mean	32731.96	2147483647.50	32767.50
Std. dev.	18935.33	1109199008.12	16924.78
One-bit ratio	0.499462	0.500000	0.500000
χ^2	2.6648	103125.0000	103125.0000

Table 8: Basic LFSR statistics computed from 100000 outputs

The good LFSR has an almost perfect one-bit ratio and a low chi-square statistic, which means that the one-dimensional histogram is close to uniform in this experiment. The bad single-tap variants also have an ideal-looking one-bit ratio, but their chi-square values are extremely large. This is the important warning: bit balance alone can look good while the sequence still visits only a small and highly structured subset of the state space.

Figure 13 shows that the good LFSR distributes values evenly between histogram buckets. The bit ratio is close to 0.5, so zeros and ones appear with nearly equal frequency in the output bits.

Figure 14 shows that the values cover the available range without immediately collapsing into a short visible cycle. Since an LFSR is linear over $GF(2)$, this still does not make it cryptographically secure, but it is acceptable as a basic demonstration of a well-chosen feedback register.

Figure 15 shows that using a large state is not enough. The bit ratio is exactly balanced, but the histogram is very poor. The generator has a large numerical range, but the chosen feedback rule produces a short structured cycle.

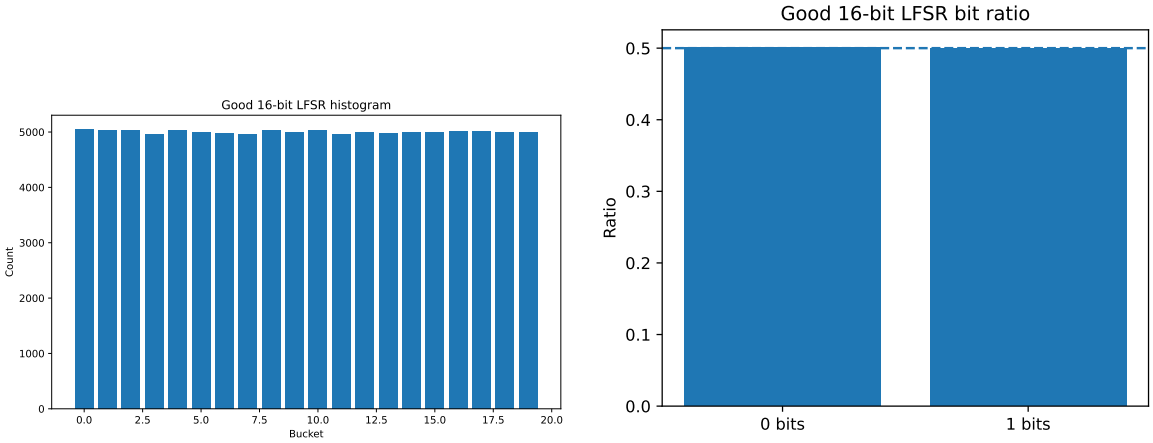

Figure 13: Histogram and bit ratio for the good LFSR

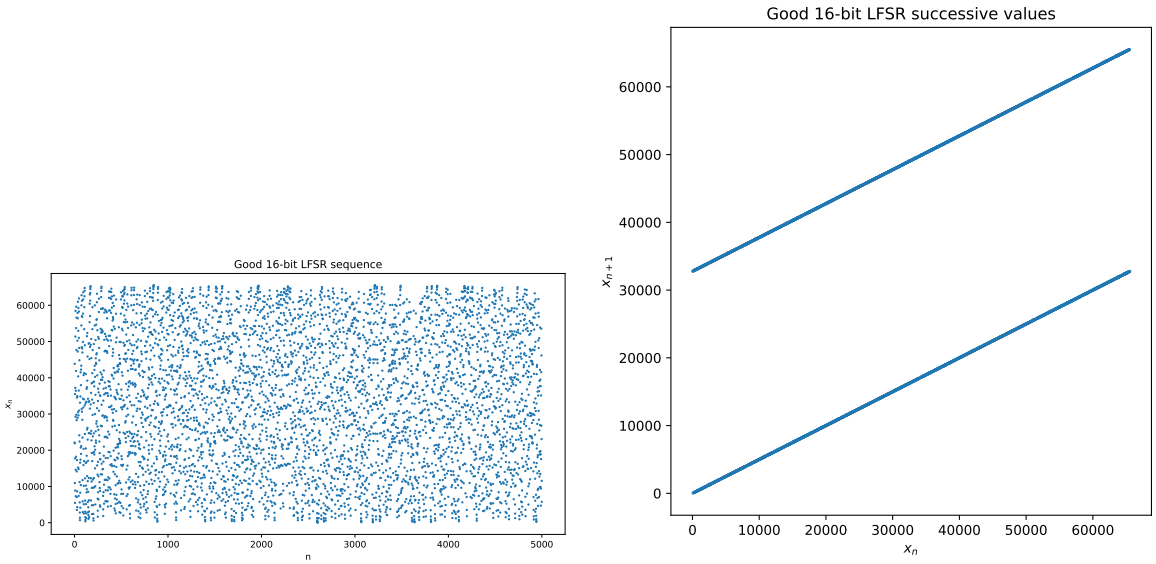

Figure 14: Sequence plot and successive-value plot for the good LFSR

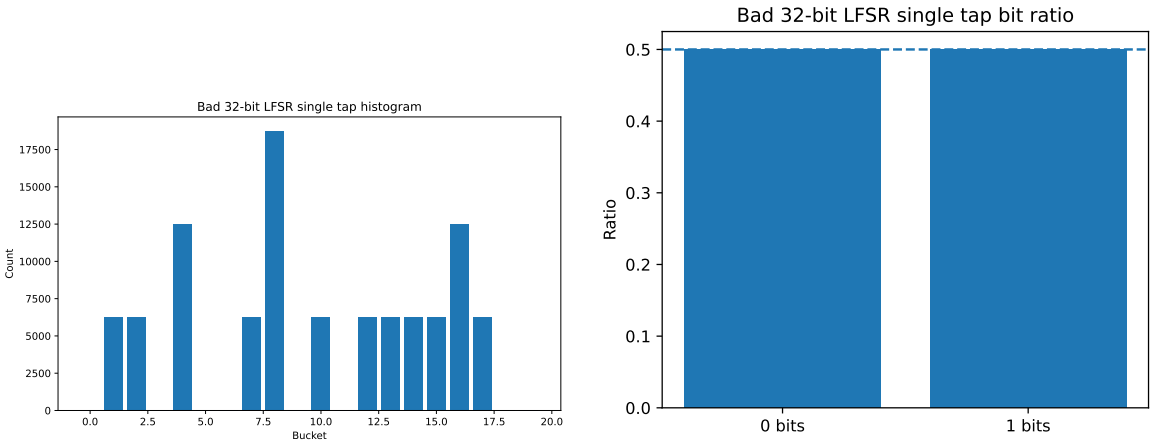

Figure 15: Histogram and bit ratio for the bad 32-bit single-tap LFSR

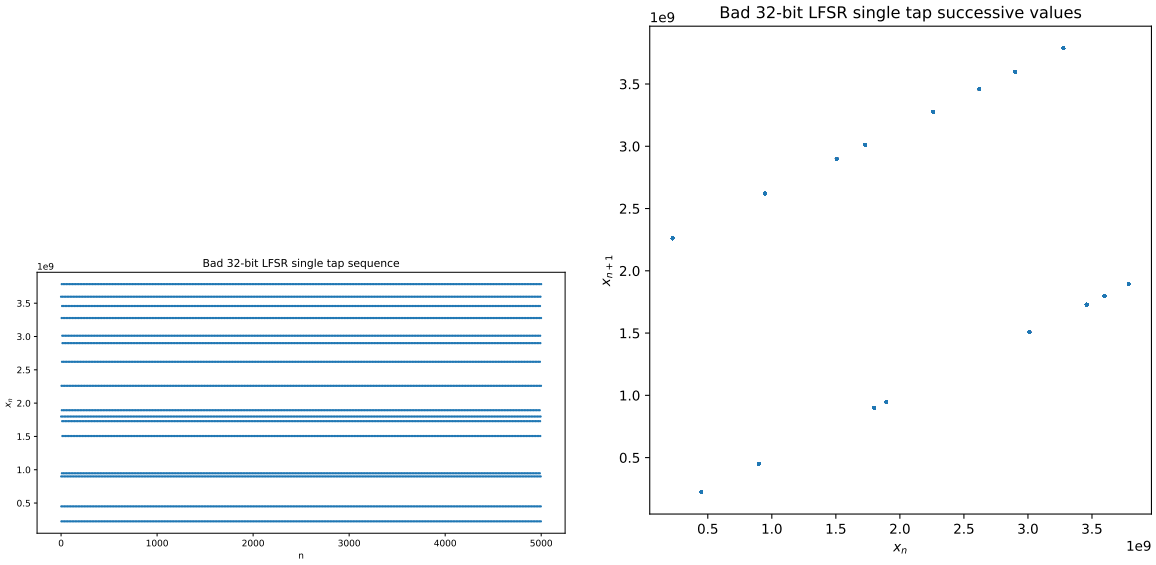

Figure 16: Sequence plot and successive-value plot for the bad 32-bit single-tap LFSR

Figure 16 makes the structural problem visible. The points do not fill the plane in a random-looking way. Instead, they follow a simple deterministic pattern caused by the poor tap choice.

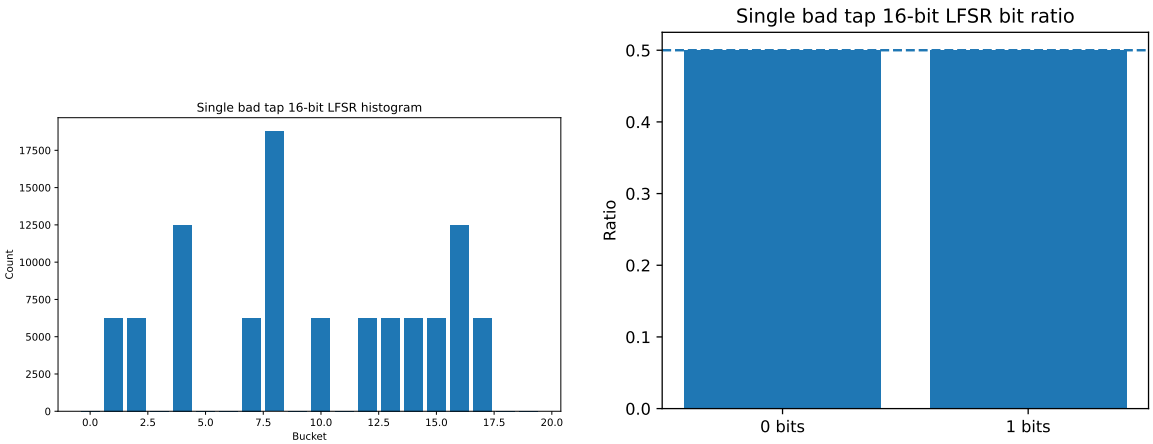

Figure 17: Histogram and bit ratio for the 16-bit LFSR with one bad tap choice

Figure 17 shows the same type of failure in a smaller register. The one-bit ratio still looks perfect, but the histogram shows that the values are not distributed correctly.

Figure 18 confirms that changing the tap set can destroy the period and create a short deterministic structure. This is why LFSR taps should be selected using primitive polynomials rather than chosen arbitrarily.

3.6 Lagged Fibonacci Generator (LFG)

A Lagged Fibonacci Generator is a recursive pseudorandom number generator that generalizes the Fibonacci recurrence. Instead of using only the two immediately previous

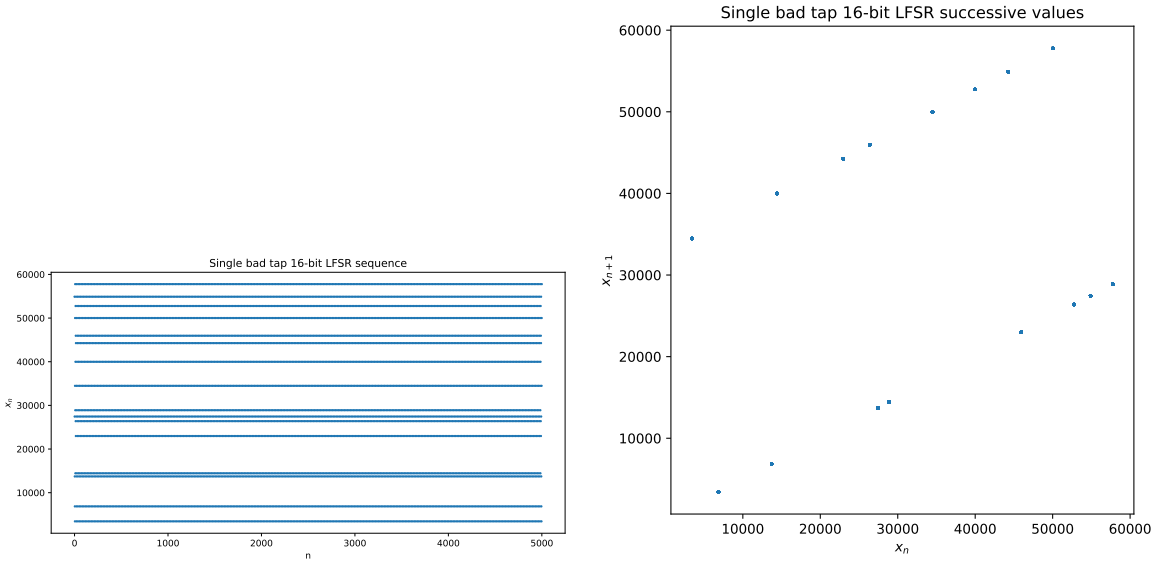

Figure 18: Sequence plot and successive-value plot for the 16-bit LFSR with one bad tap choice

values, it combines two older values separated by fixed lags. In the additive version used here, the recurrence is

$$x_n = (x_{n-j} + x_{n-k}) \bmod 2^{64},$$

where j and k are positive integers with $j < k$. The generator stores the last k values in an internal circular buffer.

The quality of an LFG depends on the choice of lags and on the initialization of the whole state array. Unlike an LCG or a simple LFSR, the seed is not just one value of the recurrence. It must be expanded into a complete initial table of values. Poor initialization may create visible patterns even when the recurrence itself has reasonable parameters.

3.7 LFG C++ Implementation

The implementation stores the state in an array of length equal to the long lag. The index points to the position that will be overwritten. In each call of `next()`, two previous values are selected from the circular buffer, added together, and the result replaces the oldest value.

Listing 5: Implementation of the Lagged Fibonacci Generator transition function

```
uint64_t next() {
    std::size_t first = (_index + _longLag - _shortLag) % _longLag;
    std::size_t second = _index;
    uint64_t value = _state[first] + _state[second];
    _state[_index] = value;
    _index = (_index + 1) % _longLag;
    return value;
```

```
}
```

The recurrence is very fast, because it only requires indexing, integer addition, and one circular-buffer update. The modulo 2^{64} operation is implicit for unsigned 64-bit arithmetic: overflow automatically wraps around. This makes the implementation simple, but the statistical quality still depends on good lags and a properly mixed initial state.

3.8 LFG Experimental Comparison

Three LFG variants were compared. The first one uses the classical additive lag pair $(24, 55)$ and a mixed initial state. The second one uses very short lags $(1, 2)$, which makes the recurrence much closer to the ordinary Fibonacci recurrence and increases dependence between consecutive values. The third one keeps the good lags but uses a constant initial state, showing how bad initialization can damage the generator.

Parameter	Good LFG	Bad short-lag LFG	Constant-state LFG
Seed	42	42	constant state
Short lag j	24	1	24
Long lag k	55	2	55
Operation	addition mod 2^{64}	addition mod 2^{64}	addition mod 2^{64}

Table 9: LFG parameter sets used in the experiment

Statistic	Good LFG	Bad short-lag LFG	Constant-state LFG
Min	$1.3171 \cdot 10^{14}$	$4.3654 \cdot 10^{14}$	2
Max	$1.8447 \cdot 10^{19}$	$1.8447 \cdot 10^{19}$	$1.8447 \cdot 10^{19}$
Mean	$9.2319 \cdot 10^{18}$	$9.2302 \cdot 10^{18}$	$9.0300 \cdot 10^{18}$
Std. dev.	$5.3384 \cdot 10^{18}$	$5.3292 \cdot 10^{18}$	$5.4468 \cdot 10^{18}$
One-bit ratio	0.499816	0.505044	0.494553
χ^2	15.4312	22.6568	977.5000

Table 10: Basic LFG statistics computed from 100000 outputs

The good LFG has a one-bit ratio close to 0.5 and a moderate chi-square statistic. The short-lag generator still covers almost the whole 64-bit range, so its minimum and maximum do not immediately reveal the problem. However, its bit ratio is shifted away from 0.5, and the recurrence has much stronger local dependence. The constant-state variant has the largest chi-square statistic, which indicates a strong distortion in the one-dimensional distribution.

Figure 19 shows that the good LFG produces an approximately balanced distribution in this basic test. The histogram buckets are close to each other, and the bit ratio is near the ideal value 0.5.

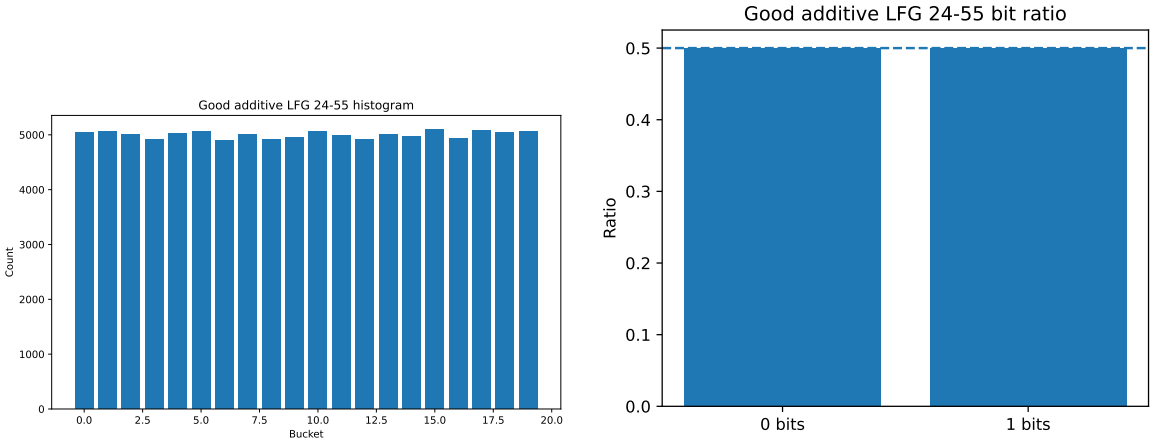

Figure 19: Histogram and bit ratio for the good LFG

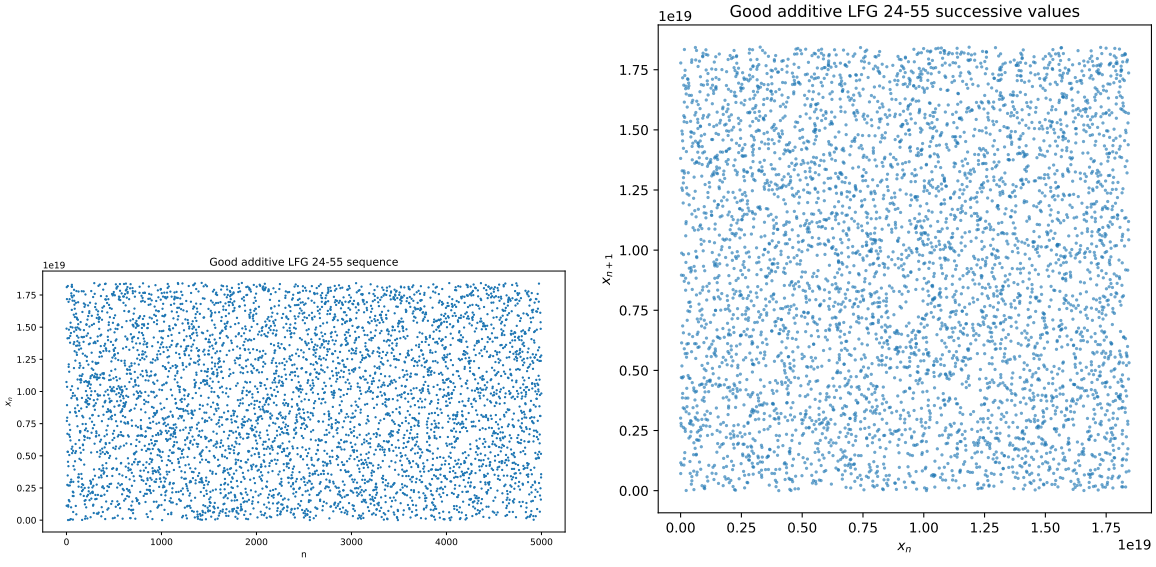

Figure 20: Sequence plot and successive-value plot for the good LFG

Figure 20 shows that the values spread across the 64-bit range. The successive-value plot does not collapse into a simple visible line, which is the expected behaviour for a reasonably initialized additive LFG.

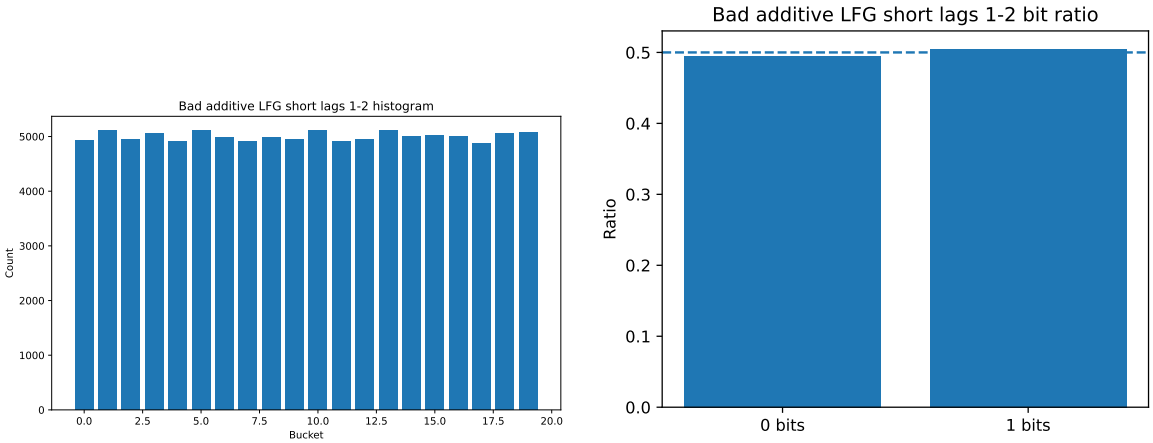

Figure 21: Histogram and bit ratio for the bad short-lag LFG

Figure 21 shows that the short-lag LFG may still look acceptable in a simple histogram, but its bit ratio is less balanced than in the good case. This illustrates that large numerical range alone does not guarantee good pseudorandom behaviour.

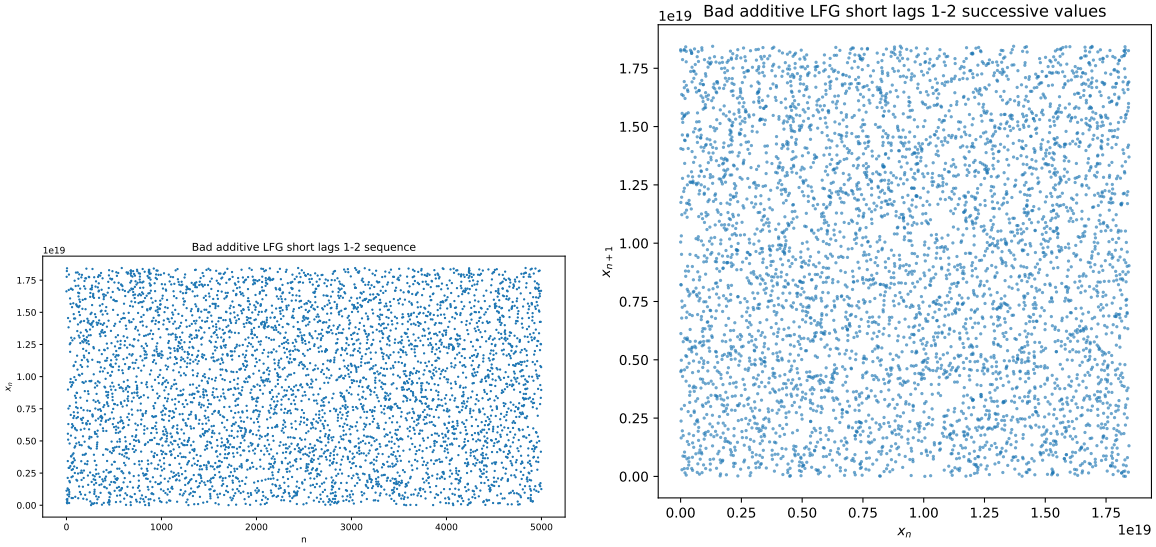

Figure 22: Sequence plot and successive-value plot for the bad short-lag LFG

Figure 22 shows stronger dependence between consecutive values than in the good case. Since the lags are too small, the generator behaves more like a simple Fibonacci recurrence than a well-separated lagged generator.

Figure 23 shows that a bad initial state can strongly damage the distribution. The chi-square statistic is much larger than in the good case, and the bit ratio is visibly shifted away from 0.5.

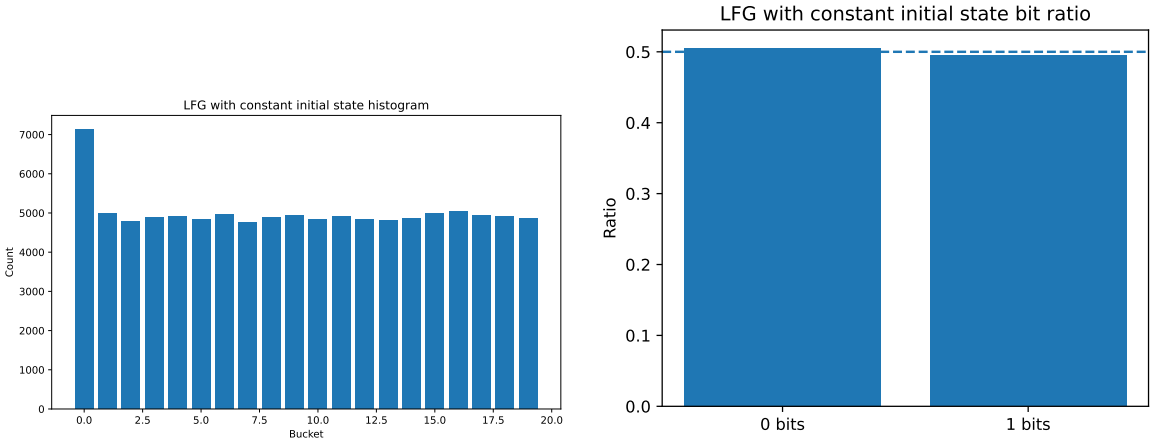

Figure 23: Histogram and bit ratio for the LFG with constant initial state

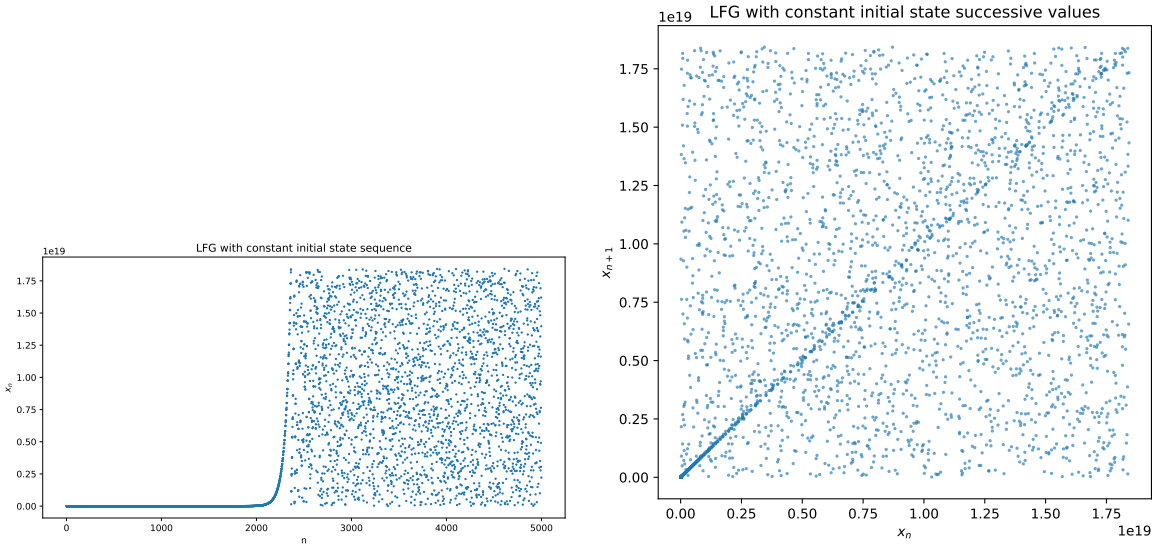

Figure 24: Sequence plot and successive-value plot for the LFG with constant initial state

Figure 24 confirms that initialization is part of the generator design. Even with reasonable lag values, starting from a poorly mixed state introduces visible structure. Therefore, LFGs should use both carefully selected lags and a non-degenerate initialization procedure.

4 The Mersenne Twister: High-Dimensional Equidistribution

Developed in 1997 by Makoto Matsumoto and Takuji Nishimura [6], the Mersenne Twister (MT) is a widely used pseudorandom number generator in modern computing, serving as the default or standard option in software environments including Python's `random` module, R, MATLAB, and the C++ standard library.

By choosing parameters such that the effective state size $nw - r$ corresponds to a Mersenne exponent, the MT algorithm achieves a staggering periodicity and an extraordinarily high degree of multidimensional equidistribution.

4.1 The Twisted Generalized Feedback Shift Register (TGFSR)

The Mersenne Twister is based on Generalized Feedback Shift Register (GFSR). While a standard LFSR operates on single bits over $\mathbb{F}_2$, a GFSR operates on w-bit words, effectively running w identical LFSRs in parallel.

The Mersenne Twister is a specific optimization of a *Twisted* Generalized Feedback Shift Register (TGFSR). Let the word size be w bits, and let the state array consist of n words. The state at step k is represented by a sequence of w-bit row vectors $\mathbf{x}_k \in \mathbb{F}_2^w$.

4.2 The Twist

The TGFSR defines the state transition via a linear recurrence relation of order n. The generator computes the subsequent word $\mathbf{x}_{k+n}$ using the word from n steps ago ($\mathbf{x}_k$) and a word from a mid-point m steps ago ($\mathbf{x}_{k+m}$), augmented by a linear transformation matrix A:

$$\mathbf{x}_{k+n} = \mathbf{x}_{k+m} \oplus \left(\mathbf{x}_k^u \mid \mathbf{x}_{k+1}^l\right) A$$

Let us rigorously dissect the components of this linear transformation over $\mathbb{F}_2$:

Concatenation (|): The term $\left(\mathbf{x}_k^u \mid \mathbf{x}_{k+1}^l\right)$ represents a bitwise concatenation. We extract the upper $w - r$ bits from $\mathbf{x}_k$ (denoted $\mathbf{x}_k^u$) and the lower r bits from the next state $\mathbf{x}_{k+1}$ (denoted $\mathbf{x}_{k+1}^l$). This creates a new, composite w-bit word. This incomplete word transition is one of the key innovations of the Mersenne Twister architecture, allowing the state space to precisely match a Mersenne prime rather than a power of 2.

The Matrix A: The concatenated word is multiplied by a $w \times w$ matrix A over $\mathbb{F}_2$. To ensure maximum computational efficiency in software, A is chosen to be in Rational

Normal Form (a companion matrix):

$$A = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ a_{w-1} & a_{w-2} & a_{w-3} & \cdots & a_0 \end{pmatrix}$$

Remark. *Because A is in rational normal form, multiplying a vector $\mathbf{y} = (y_{w-1}, \ldots, y_0)$ by A can be executed via a single bit-shift and a conditional XOR. If the lowest bit $y_0 = 0$, the result is $\mathbf{y} \gg 1$. If $y_0 = 1$, the result is $(\mathbf{y} \gg 1) \oplus \mathbf{a}$, where $\mathbf{a} = (a_{w-1}, \ldots, a_0)$.*

4.3 Tempering: Guaranteeing Equidistribution

To avoid high degrees of linear correlation, failing rigorous tests for k-distribution, we add **tempering** to the process, which is an invertible linear transformation over $\mathbb{F}_2$. The tempering function $T : \mathbb{F}_2^w \to \mathbb{F}_2^w$ is designed to multiply the state vector by an invertible matrix T, which systematically scrambles the bits to achieve an optimal multidimensional distribution.

Given a raw state word $\mathbf{x}$, the tempered output $\mathbf{y}$ is generated through four successive shift-and-mask operations, utilizing shifting constants (u, s, t, l) and bitmask vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}$:

$$\begin{aligned} \mathbf{y}_1 &= \mathbf{x} \oplus ((\mathbf{x} \gg u) \,\&\, \mathbf{a}) \\ \mathbf{y}_2 &= \mathbf{y}_1 \oplus ((\mathbf{y}_1 \ll s) \,\&\, \mathbf{b}) \\ \mathbf{y}_3 &= \mathbf{y}_2 \oplus ((\mathbf{y}_2 \ll t) \,\&\, \mathbf{c}) \\ \mathbf{y} &= \mathbf{y}_3 \oplus ((\mathbf{y}_3 \gg l) \,\&\, \mathbf{d}) \end{aligned}$$

where & denotes the bitwise AND operation.

Theorem 4.1. *Because each step of the tempering process is an invertible linear transformation over $\mathbb{F}_2$, the composition of these transformations is likewise invertible. Hence, the entire tempering process defines a bijection on $\mathbb{F}_2^w$.*

4.4 Periodicity and Mersenne Primes

The defining characteristic of the Mersenne Twister is its periodicity. The total number of bits in the internal state is $n \times w$. However, because r bits are masked out during the concatenation step $\left(\mathbf{x}_k^u \mid \mathbf{x}_{k+1}^l\right)$, the actual operational state space consists of $nw - r$ bits.

Therefore, the maximum possible period of the generator is $2^{nw-r} - 1$.

Definition 4.2. *A **Mersenne prime** is a prime number that can be written in the form $M_p = 2^p - 1$, where p is a prime integer.*

By carefully choosing the parameters n, w, and r such that $p = nw - r$ corresponds to the exponent of a Mersenne prime, Matsumoto and Nishimura simplified the mathematical analysis required to construct a primitive characteristic polynomial for the generator.

4.5 Variants of the Mersenne Twister

4.5.1 The MT19937 Variant

The most popular configuration of the Mersenne Twister is MT19937 [6]. The parameters are chosen as follows:

- Word size $w = 32$ bits.
- Degree of recurrence $n = 624$ words.
- Separation point $m = 397$.
- Lower bitmask size $r = 31$ bits.

This configuration results in a total operational state size of $624 \times 32 - 31 = 19937$ bits. The parameters of MT19937 were specifically selected so that the associated characteristic polynomial is primitive over $\mathbb{F}_2$, yielding the maximal possible period

$$2^{19937} - 1.$$

The fact that 19937 is a Mersenne exponent plays an important role in the underlying construction and analysis.

4.5.2 Other variants of the MT

1. **CryptMT:** CryptMT is a stream cipher and cryptographically secure pseudorandom number generator (CSPRNG). It was developed by Makoto Matsumoto and Takuji Nishimura alongside Mariko Hagita and Mutsuo Saito. Unlike the classical Mersenne Twister or its other structural derivatives, CryptMT is patented.
2. **MTGP:** MTGP (Mersenne Twister for Graphics Processors) is a variant [7] optimized for graphics processing units published by Mutsuo Saito and Makoto Matsumoto. The basic linear recurrence operations are extended from MT, and parameters are carefully chosen to allow a massive pool of parallel threads to compute the recursion concurrently while sharing their active state space to minimize memory load. The original paper claims improved multidimensional equidistribution over

standard MT, displaying execution metrics on a 2008-era GPU (Nvidia GTX260 with 192 cores) of 4.7 ms to produce 5×10^7 random 32-bit integers.

3. **SFMT:** SFMT (SIMD-oriented Fast Mersenne Twister) is a variant introduced in 2006 designed specifically to leverage speed advantages when executing on 128-bit SIMD architectures. It is roughly twice as fast as the classical Mersenne Twister.

4. **TinyMT:** TinyMT is a lightweight variant proposed by Saito and Matsumoto in 2011. It utilizes just 127 bits of internal state space, marking a significant footprint reduction compared to the 2.5 KiB state array required by the classic MT19937. However, its maximum period is bounded at:

$$2^{127} - 1$$

4.6 C++ Implementation

The core output operation of the Mersenne Twister is implemented in the `next()` method. When all words from the current state array have already been consumed, the generator first performs the twist step and rebuilds the internal state. Then it takes one raw 32-bit word from the state array and applies tempering.

Listing 6: Output function of MT19937

```
uint32_t next() {
    if (_index >= _state.size()) {
        twist();
    }

    uint32_t y = _state[_index++];

    y ^= (y >> 11);
    y ^= (y << 7) & 0x9D2C5680U;
    y ^= (y << 15) & 0xEFC60000U;
    y ^= (y >> 18);

    return y;
}
```

The first part of the method checks whether the internal state array must be regenerated. This corresponds to applying the TGFSR recurrence described earlier. The variable `y` is then taken from the state array and transformed by four bitwise operations. These operations are the tempering step: right shifts, left shifts, XORs and bit masks are used to improve the distribution of output bits.

The important point is that the state transition and the output transformation are separated. The twist changes the internal state, while tempering changes only the returned value. Because tempering is invertible over $\mathbb{F}_2$, it improves the statistical quality of the visible output without changing the period of the underlying state cycle.

For comparison, two modified variants were also tested. The first one removes tempering and returns the raw state word directly.

Listing 7: MT19937 output without tempering

```
uint32_t next() {
    if (_index >= _state.size()) {
        twist();
    }

    return _state[_index++];
}
```

The second modified variant removes the matrix constant from the twist operation. This keeps the output range large, but weakens the linear recurrence itself.

Listing 8: MT-like output using a weakened twist step

```
uint32_t next() {
    if (_index >= _state.size()) {
        twistWithoutMatrixA();
    }

    return _state[_index++];
}
```

4.7 Experimental Comparison

To illustrate the role of tempering and the twist matrix, three variants were compared. The first one is the standard MT19937 generator. The second one uses the same transition, but omits tempering. The third one uses a weakened twist step by removing the matrix constant A.

For each generator, 100000 generated values were used to compute basic statistics. The histogram was divided into 20 buckets. The sequence plot and the successive-value plot were generated using 5000 values.

The standard MT19937 has a mean close to the middle of the 32-bit output range and a one-bit ratio very close to 0.5. This indicates good balance between zeros and ones in the binary representation of generated values. The chi-square statistic for the histogram

Parameter	Standard MT19937	No tempering	Bad twist matrix
Seed	5489	5489	5489
Word size w	32	32	32
State size n	624	624	624
Middle word m	397	397	397
Lower mask bits r	31	31	31
Matrix constant A	`0x9908B0DF`	`0x9908B0DF`	`0x00000000`
Tempering	yes	no	no

Table 11: Mersenne Twister variants used in the experiment

Statistic	Standard MT19937	No tempering	Bad twist matrix
Min	52150	51092	1008367
Max	4294877384	4294911316	4294782864
Mean	2143446744.27	2148531592.69	2199899818.70
Std. dev.	1239920307.25	1241945764.29	1232324944.58
One-bit ratio	0.499926	0.500359	0.500918
χ^2	23.4012	20.4104	1512.2160

Table 12: Basic statistics computed from 100000 outputs

is also moderate, so the one-dimensional distribution looks close to uniform in this simple test.

The variant without tempering still looks acceptable in basic one-dimensional statistics. Its mean, standard deviation, bit ratio and histogram statistic are close to the standard version. This is expected, because removing tempering does not immediately destroy the period of the state transition. However, it exposes the raw linear structure of the state more directly, which is exactly what tempering is designed to hide.

The variant with the weakened twist matrix behaves much worse. Its chi-square statistic is much larger, even though the output range is still large. This shows that preserving a large numerical range is not enough. If the recurrence relation is weakened, the generator may produce visibly non-uniform bucket frequencies and stronger structural artifacts.

4.8 Plots and Interpretation

Figure 25 shows that the standard MT19937 distributes values evenly across the histogram buckets. The bit-ratio plot is almost exactly centered at 0.5, which confirms that zeros and ones occur with nearly equal frequency.

Figure 26 shows that the generated values cover the full 32-bit range. The successive-value plot does not collapse into a small set of visible lines, which is consistent with the high-dimensional equidistribution properties of MT19937.

Figure 27 shows that removing tempering does not necessarily create an obvious failure

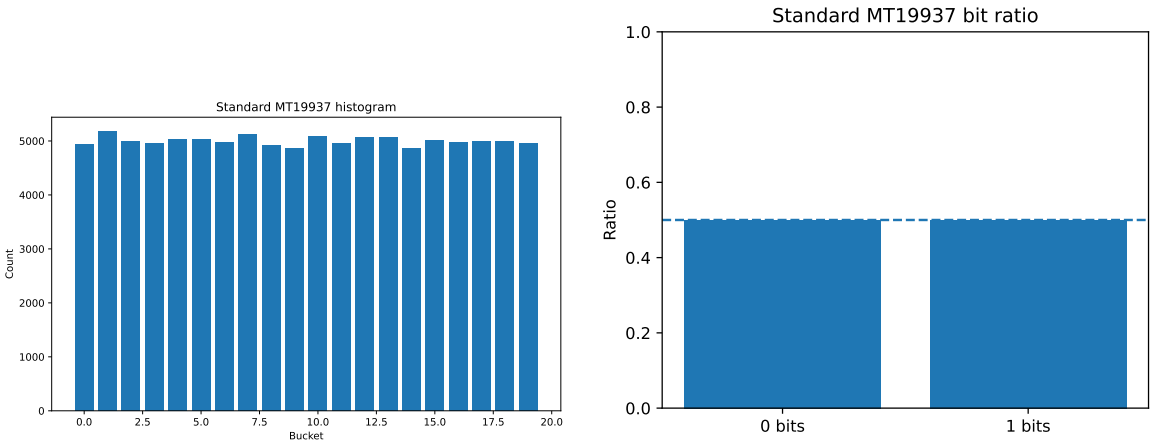

Figure 25: Histogram and bit ratio for the standard MT19937

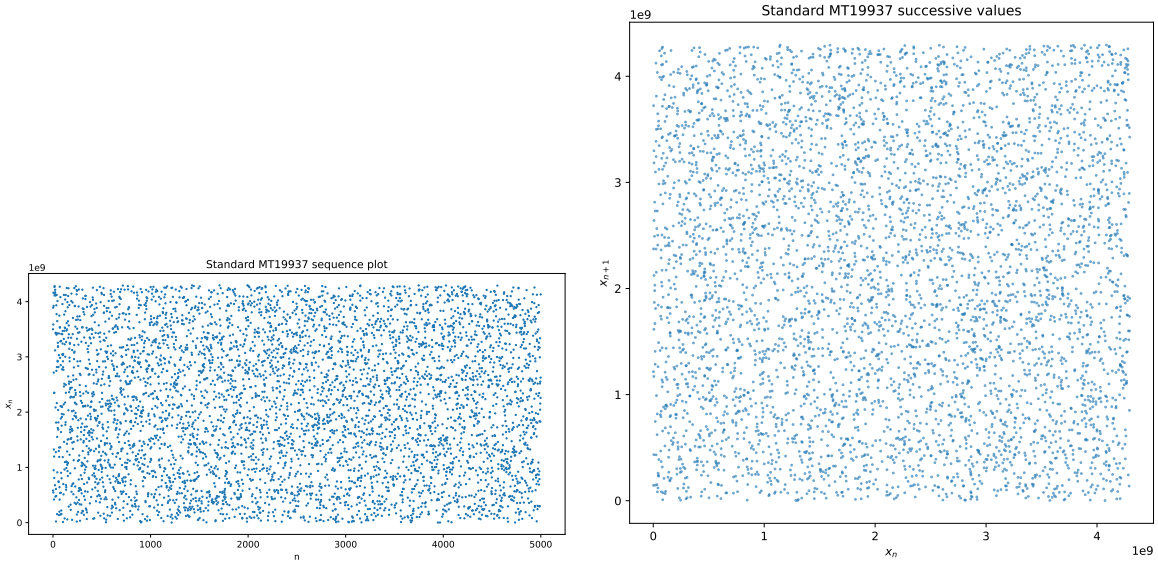

Figure 26: Sequence plot and successive-value plot for the standard MT19937

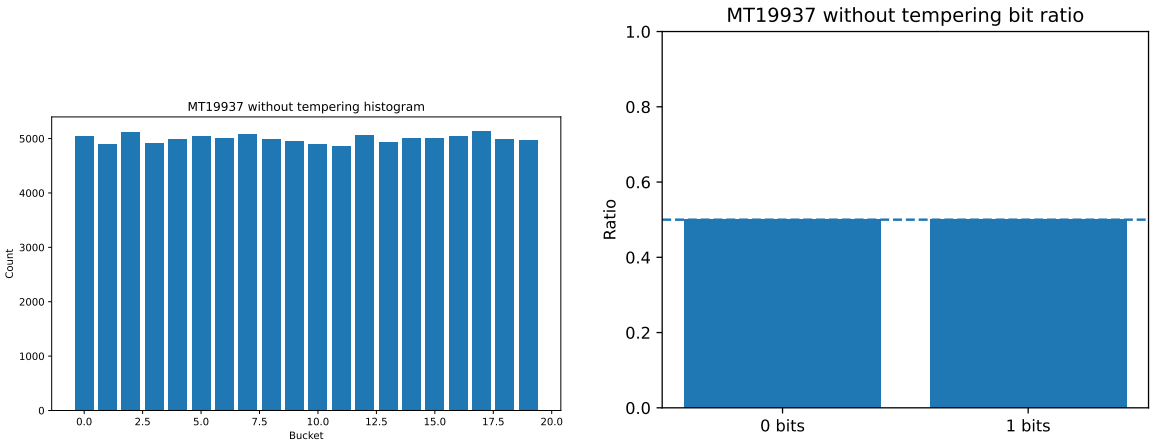

Figure 27: Histogram and bit ratio for MT19937 without tempering

in a simple histogram. This is a useful example of a generator that may look acceptable in basic tests, while still being theoretically weaker because the raw linear state is exposed.

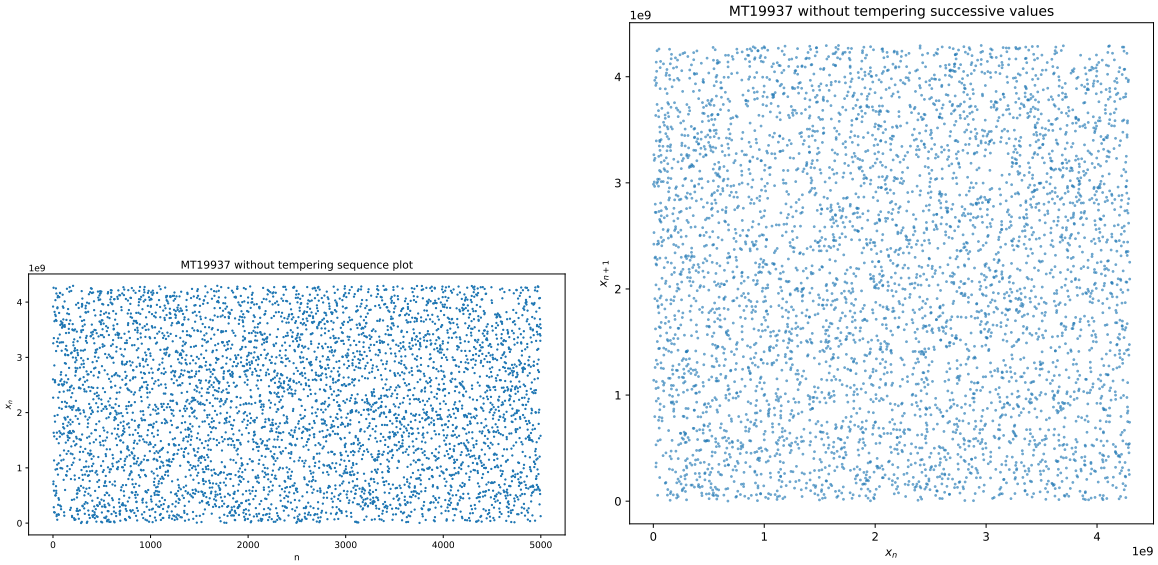

Figure 28: Sequence plot and successive-value plot for MT19937 without tempering

Figure 28 again shows that short visual tests are limited. The values still cover a broad range, but without tempering the output is a more direct observation of the internal linear recurrence.

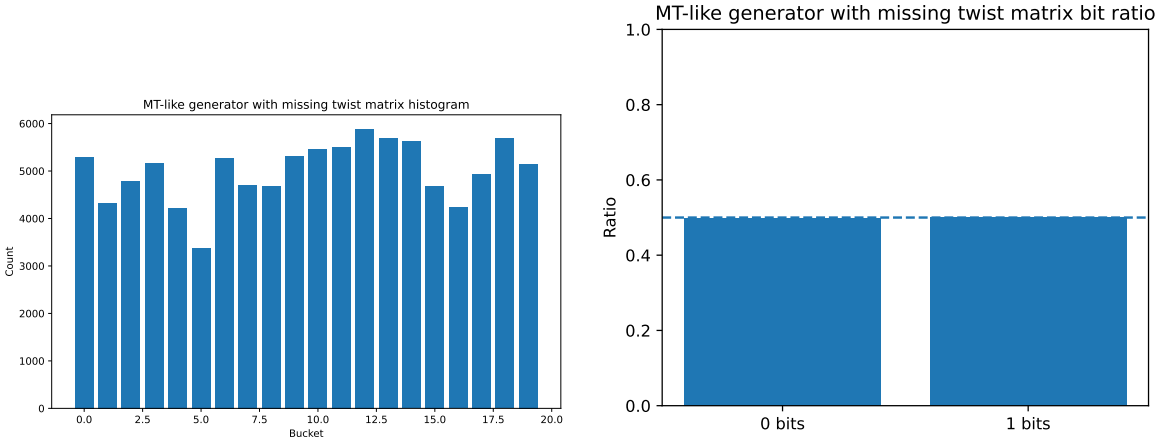

Figure 29: Histogram and bit ratio for the MT-like generator with weakened twist

Figure 29 shows the clearest failure. The bit ratio is still close to 0.5, but the histogram is visibly less uniform. This demonstrates that bit balance alone is not sufficient to judge the quality of a pseudorandom generator.

Figure 30 illustrates that the weakened recurrence still produces values from a large range, but the distributional behavior is worse. The comparison confirms that both parts of the Mersenne Twister construction are important: the twist step determines the quality of the state transition, while tempering improves the visible output distribution.

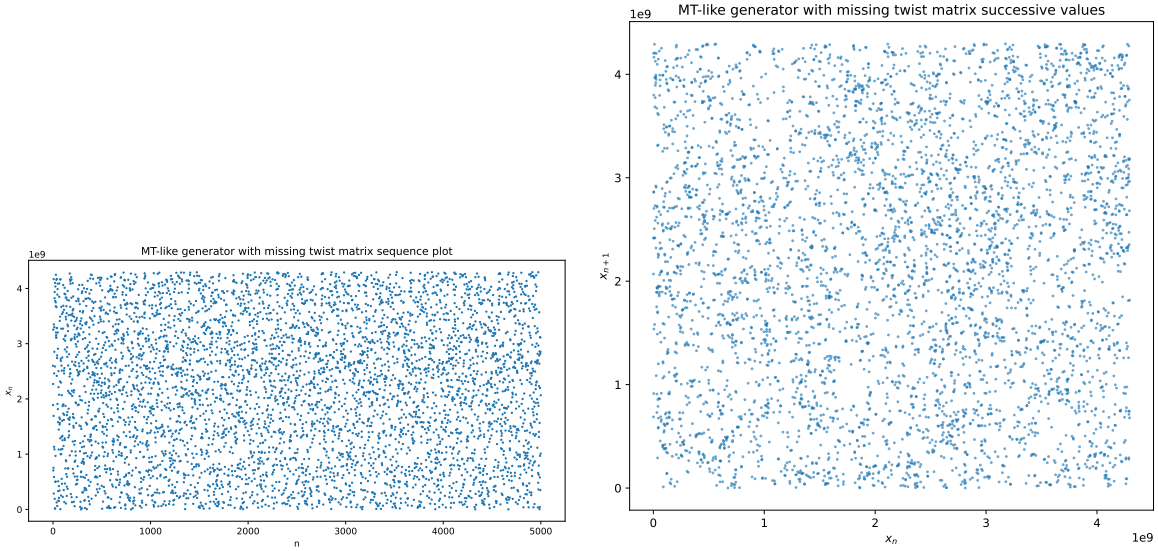

Figure 30: Sequence plot and successive-value plot for the MT-like generator with weakened twist

Overall, the experiment shows that MT19937 works well not because of a single operation, but because its parameters, recurrence relation and tempering transformation are carefully chosen together. Removing one part may not always break simple statistics immediately, but it weakens the structural properties that make the generator reliable in larger simulations.

5 Modern Pseudorandom Number Generators

While LFSRs and LFGs provide foundational algebraic methods for generating pseudorandom sequences, modern applications require generators that execute with minimal latency in software while exhibiting strong empirical statistical properties. This chapter explores two highly efficient families of generators: **Xorshift** (and its scrambled variants) and **PCG** (Permuted Congruential Generator).

5.1 The Xorshift

Introduced by George Marsaglia in 2003 [8], the Xorshift generator operates exclusively via hardware-efficient logical shift and bitwise XOR operations. Mathematically, it implements a linear transformation over the vector space $\mathbb{F}_2^w$, making it a software-efficient analogue of the LFSR.

5.1.1 The Recurrence Relation

Given a non-zero w-bit integer state $\mathbf{x}_n$, the subsequent state is generated by applying three successive shift-and-XOR operations using integer constants a, b, and c:

$$\begin{aligned}\mathbf{y}_1 &= \mathbf{x}_0 \oplus (\mathbf{x}_0 \ll a)\\ \mathbf{y}_2 &= \mathbf{y}_1 \oplus (\mathbf{y}_1 \gg b)\\ \mathbf{x}_1 &= \mathbf{y}_2 \oplus (\mathbf{y}_2 \ll c)\end{aligned}$$

5.1.2 Matrix Representation and Periodicity

Rigorously, a w-bit word is a vector in $\mathbb{F}_2^w$. A left shift by a positions equates to multiplying the state vector by a nilpotent $w \times w$ matrix L^a, where L possesses ones on the first subdiagonal. Similarly, a right shift by b is multiplication by $R^b = (L^T)^b$. Since XOR acts as standard vector addition over $\mathbb{F}_2$, the global state transition is governed by the linear map $\mathbf{x}_{n+1} = M\mathbf{x}_n$, where the transformation matrix M is:

$$M = (I + L^c)(I + R^b)(I + L^a)$$

Theorem 5.1. *An Xorshift generator defined by the state transition matrix M over $\mathbb{F}_2^w$ has a maximal cycle length (period) of $2^w - 1$ if and only if M has multiplicative order $2^w - 1$, equivalently if its minimal polynomial is primitive of degree w.*

When the minimal polynomial is primitive, the state space partitions into exactly two cycles: a trivial fixed point at the zero vector (length 1), and a single maximum-length cycle containing all $2^w - 1$ non-zero states.

5.1.3 Variants of the Xorshift

Despite achieving maximum periodicity, pure linear Xorshift generators fail rigorous empirical statistical tests (e.g., *MatrixRank* in TestU01). Because the transition function is linear over $\mathbb{F}_2$, consecutive outputs exhibit deterministically bounded matrix ranks and low-weight artifacts. To resolve this, "scrambled" variants append a non-linear output permutation to the linear state transition.

1. **Xorshift*:** Appends a modular multiplication scrambler, $y_n \equiv (\mathbf{x}_{n+1} \cdot M_*) \pmod{2^w}$, to the linear state transition. This introduces non-linearity, significantly improving empirical statistical quality.

2. **Xorshift+:** Expands the state space to multiple words and computes the output via modular addition: $y_n \equiv (\mathbf{s}_0 + \mathbf{s}_1) \pmod{2^w}$. However, because the initial carry-in is zero, the least significant bit remains strictly linear ($s_{0,0} \oplus s_{1,0}$), causing it to fail bit-reversed statistical tests.

3. **Xoshiro (xor, shift, rotate) / Xoroshiro (xor, rotate, shift, rotate):** Integrates circular rotations into transition for faster bit diffusion across 128-bit or 256-bit states. It eliminates the lower-bit linearity flaw of Xorshift+ by utilizing dual-domain scramblers.

5.1.4 C++ Implementation of Xorshift

The implementation of Xorshift follows the recurrence described above. The state is modified in-place by three shift-and-XOR operations. Each operation mixes bits from different positions, and the final state is returned as the output.

Listing 9: Implementation of the Xorshift32 `next()` function

```
uint32_t next() {
    _state ^= _state << _shiftA;
    _state ^= _state >> _shiftB;
    _state ^= _state << _shiftC;
    return _state;
}
```

A scrambled variant, Xorshift*, uses the same type of linear state transition but multiplies the final state by a fixed odd constant. This multiplication is not linear over $\mathbb{F}_2$ and therefore improves the visible output quality.

Listing 10: Implementation of the Xorshift* `next()` function

```
uint64_t next() {
    _state ^= _state >> 12;
```

```
3       _state ^= _state << 25;
4       _state ^= _state >> 27;
5       return _state * _multiplier;
6   }
```

5.1.5 Experimental Comparison of Xorshift

Three variants were tested. The first one uses the classical shift parameters $(13, 17, 5)$. The second one uses a non-standard set of shifts, still over a large 32-bit range, but with weaker mixing. The third one changes only one shift parameter: the last shift is changed from 5 to 1. This keeps the generator non-trivial and still operating over the full 32-bit range, but it weakens the bit mixing.

Variant	Seed	Shift a	Shift b	Shift c
Good Xorshift32	2463534242	13	17	5
Bad large-range Xorshift32	2463534242	31	1	31
One weak-shift Xorshift32	2463534242	13	17	1

Table 13: Xorshift parameter sets used in the experiment

Variant	Min	Max	Mean	Std. dev.	One-bit ratio	χ^2
Good Xorshift32	95953	4294949870	2149746614.22	1242030056.19	0.500326	21.6532
Bad large-range Xorshift32	469467027	3988719432	2426824957.03	1045875667.31	0.519531	130468.7500
One weak-shift Xorshift32	89687	4294910162	2144856443.53	1238218476.87	0.499973	19.0068

Table 14: Basic Xorshift statistics computed from 100000 outputs

The good Xorshift32 variant has a mean close to the middle of the 32-bit output range and a one-bit ratio close to 0.5. The histogram chi-square value is also moderate, which suggests that the one-dimensional distribution is reasonably balanced in this simple test.

The bad large-range variant is not a toy constant example: it still operates on 32-bit values. However, the shift parameters $(31, 1, 31)$ mix bits poorly. This is visible in the histogram and in the bit ratio, which is noticeably shifted away from the ideal value 0.5.

The one weak-shift variant changes only the last shift from 5 to 1. The basic statistics remain close to the good generator, which shows that a single simple experiment may fail to expose a weaker parameter choice. This case is useful as a warning: not every bad parameter choice produces an immediately absurd plot.

Figure 31 shows that the good variant distributes values relatively evenly across buckets and keeps the bit ratio close to the ideal value 0.5.

Figure 32 shows that the generated values cover the output range. However, this does not prove full statistical quality, because pure Xorshift remains linear over $\mathbb{F}_2$.

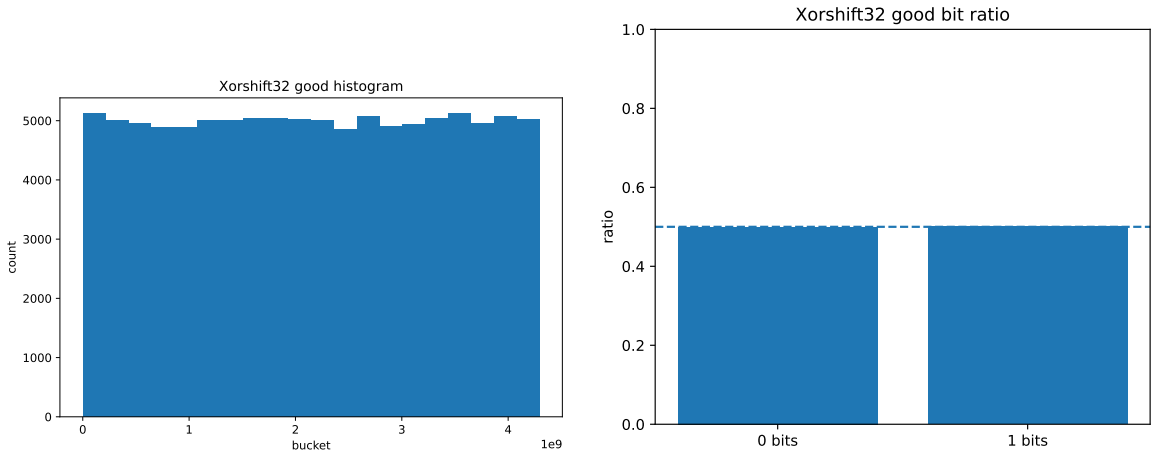

Figure 31: Histogram and bit ratio for the good Xorshift32 generator

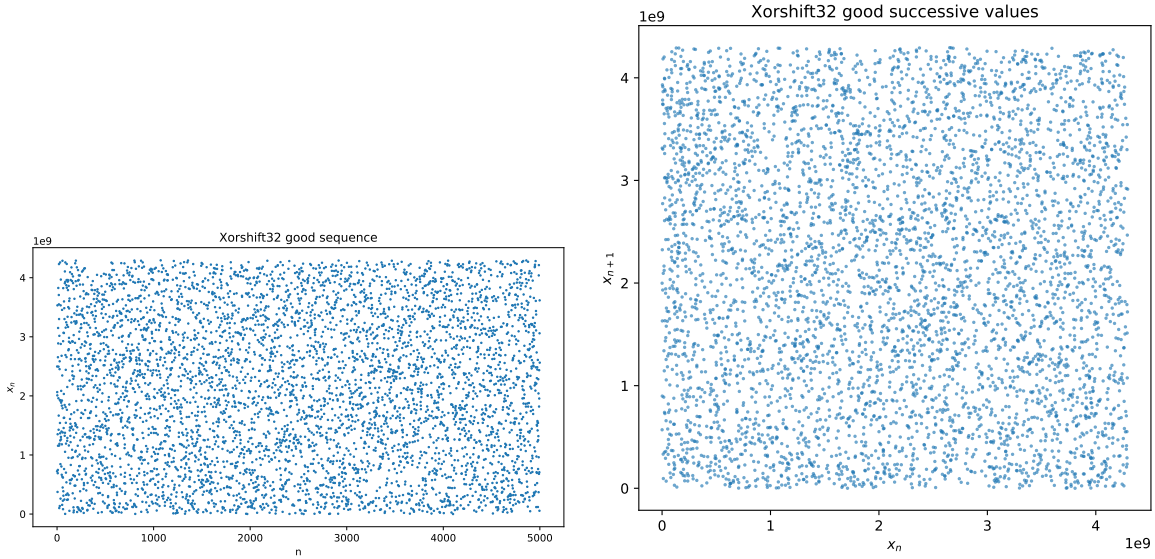

Figure 32: Sequence plot and successive-value plot for the good Xorshift32 generator

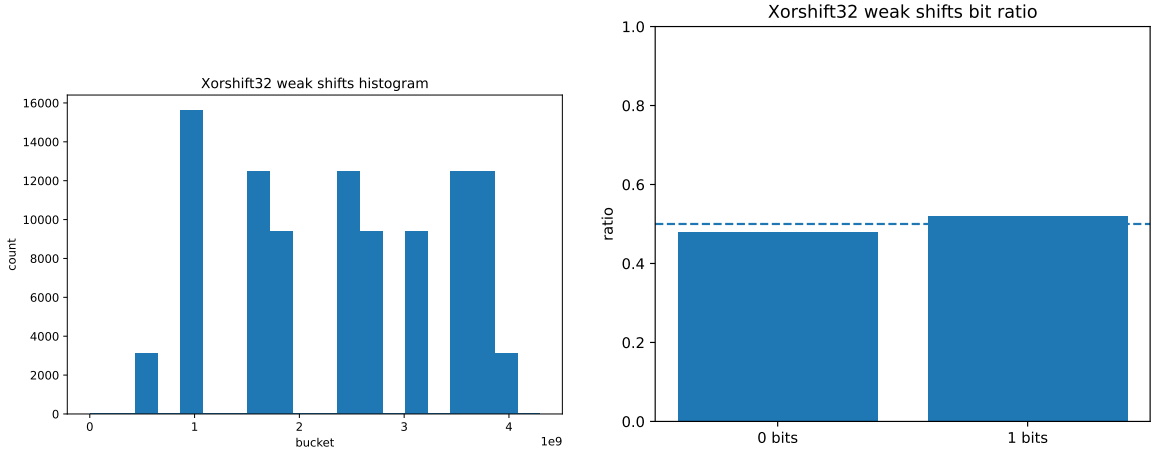

Figure 33: Histogram and bit ratio for the bad large-range Xorshift32 generator

Figure 33 shows the weaker $(31, 1, 31)$ shift choice. Unlike the zero-state example, this generator still produces values over a large range, but the bucket distribution and bit ratio are visibly worse than for the standard parameters.

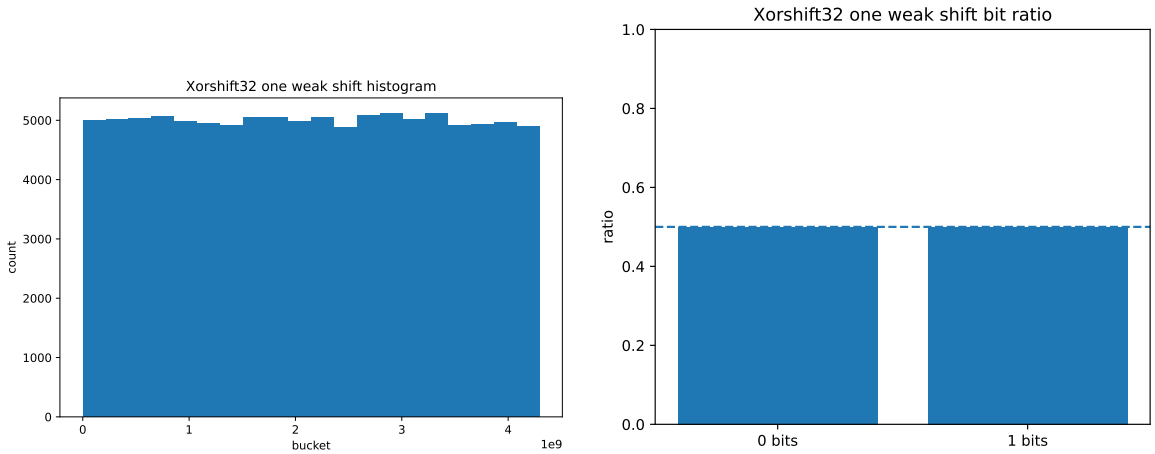

Figure 34: Histogram and bit ratio for the one weak-shift Xorshift32 generator

Figure 34 shows the case where only one shift parameter was changed. The plot does not collapse, but this is precisely the point: a visually acceptable short run is not enough to prove that the selected transition matrix has maximal period or good empirical behavior.

5.2 The PCG

Introduced by Melissa E. O'Neill in 2014 [9], the PCG framework operates exclusively via hardware-efficient modular arithmetic and bitwise permutation operations. Mathematically, it implements a non-linear transformation over the ring $\mathbb{Z}/2^k\mathbb{Z}$, making it an empirically stronger alternative to classical LCG.

5.2.1 Representation and Periodicity

The core engine of PCG is an affine recurrence relation operating over the integer ring $\mathbb{Z}/2^k\mathbb{Z}$:

$$S_{n+1} \equiv (aS_n + c) \pmod{2^k}$$

By choosing a multiplier $a \equiv 1 \pmod 4$ and an odd increment c, the generator guarantees an unbroken maximal cycle of length 2^k.

Crucially, the increment c uniquely defines the *stream* of the generator. Because any odd c yields a full period, a k-bit state space allows for exactly 2^{k-1} distinct streams. This cycle topology permits developers to assign distinct streams (via different constants c) to parallel threads, mathematically guaranteeing distinct state trajectories for different streams.

5.2.2 Popular variants of the PCG Framework

Despite achieving maximum periodicity, pure linear congruential generators fail rigorous empirical statistical tests (e.g., *Spectral Test* in TestU01). To resolve this issue, "permuted" variants append a non-linear output permutation to the affine state transition.

1. **PCG-XSH-RR:** Uses a random rotation scrambler, $u_n = \text{ROTR}(X, r)$. This introduces non-linear bit diffusion, significantly improving empirical statistical quality.
2. **PCG-XSH-RS:** Reduces the permutation operations to a simple random shift and computes the output via logical right shifts: $u_n = X \gg r$. However, because it avoids circular rotation, the lowest bits remain partially correlated to the state, causing it to fail some bit-reversed statistical tests.
3. **PCG-RXS-M-XS ("random xorshift, multiply, xorshift"):** Integrates an internal 64-bit multiplication into the permutation for faster bit diffusion across 32-bit or 64-bit states.

5.2.3 C++ Implementation of PCG

The PCG generator combines an LCG-style internal state transition with an output permutation. First, the internal 64-bit state is advanced using an affine recurrence. Then the old state is transformed using xorshift and rotation operations to produce a 32-bit output.

Listing 11: Implementation of the PCG32 `next()` function

```
uint32_t next() {
    uint64_t oldState = _state;
    _state = oldState * _multiplier + _increment;

    uint32_t xorshifted = static_cast<uint32_t>(
        ((oldState >> 18U) ^ oldState) >> 27U
    );

    uint32_t rotation = static_cast<uint32_t>(oldState >> 59U);

    return (xorshifted >> rotation) |
           (xorshifted << ((-rotation) & 31));
}
```

This separation between the state transition and the output function is the main idea of PCG. The internal transition gives a long period when the multiplier and increment satisfy the required conditions, while the permutation hides the linear structure of the underlying congruential sequence.

5.2.4 Experimental Comparison of PCG

Three PCG variants were tested. The first one uses the standard multiplier and an odd increment, so it satisfies the full-period condition for the affine recurrence modulo 2^{64}. The second one uses an even increment. The third one uses zero increment, turning the transition into a purely multiplicative recurrence.

Variant	Seed	Multiplier	Increment
Good PCG32	42	6364136223846793005	109
Bad large-range PCG32	42	6364136223846793005	54
Zero-increment PCG32	42	6364136223846793005	0

Table 15: PCG parameter sets used in the experiment

Variant	Min	Max	Mean	Std. dev.	One-bit ratio	χ^2
Good PCG32	0	4294958997	2144931975.79	1239128951.68	0.499635	16.2216
Bad large-range PCG32	0	4294960179	2146754647.86	1238184206.83	0.500030	26.4868
Zero-increment PCG32	0	4294954284	2147657129.78	1239210617.46	0.499794	12.4160

Table 16: Basic PCG statistics computed from 100000 outputs

The good PCG32 variant has a balanced bit ratio and a histogram close to uniform in this experiment. The output permutation makes the generator look much less directly tied to the linear congruential state transition.

The even-increment and zero-increment variants still produce values over a large range in a short test, so their histograms may look deceptively acceptable. However, they violate the usual full-period requirement that the increment should be odd. Therefore, their problem is not necessarily visible in every short plot, but it is structural.

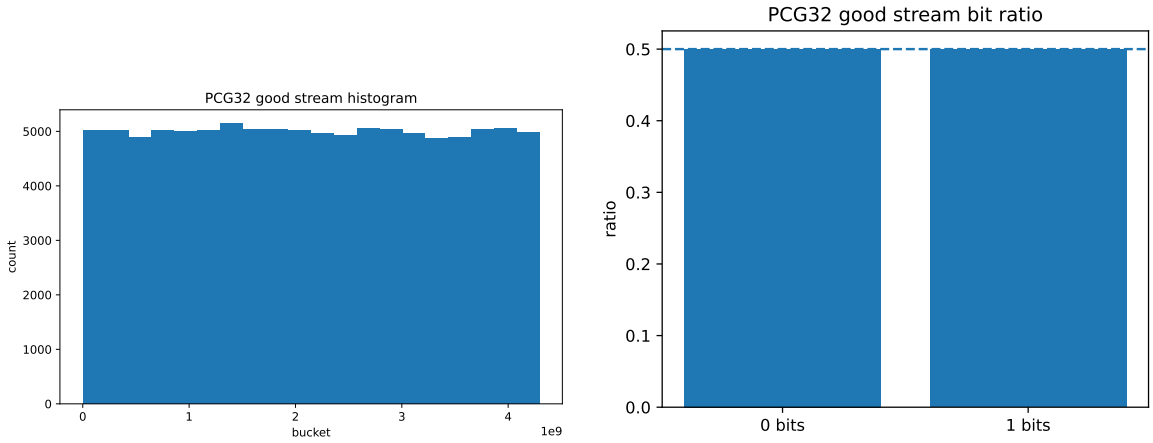

Figure 35: Histogram and bit ratio for the good PCG32 generator

Figure 35 shows a balanced one-dimensional distribution and a bit ratio close to 0.5.

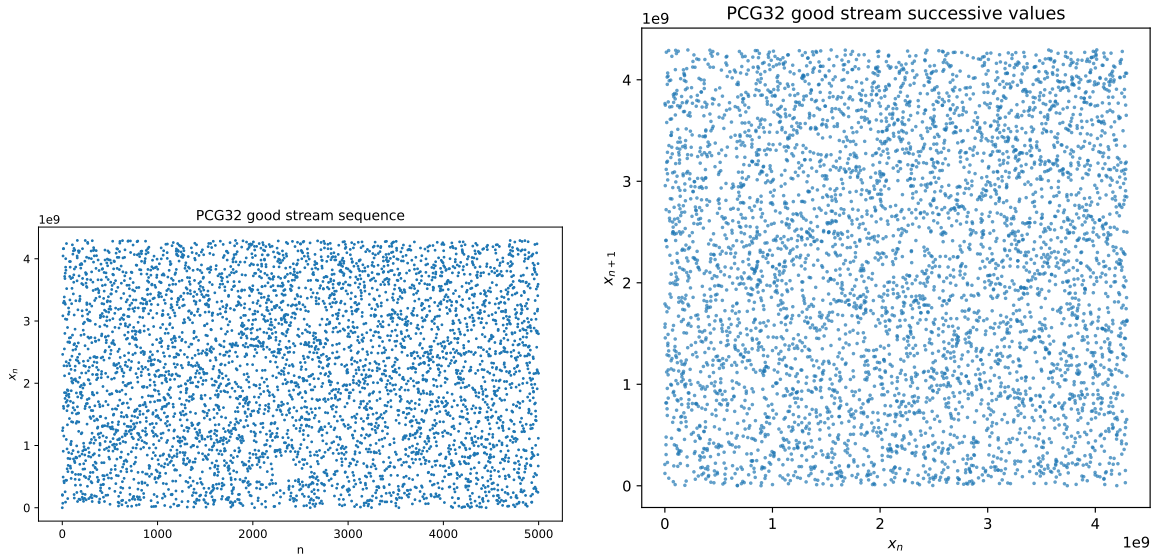

Figure 36: Sequence plot and successive-value plot for the good PCG32 generator

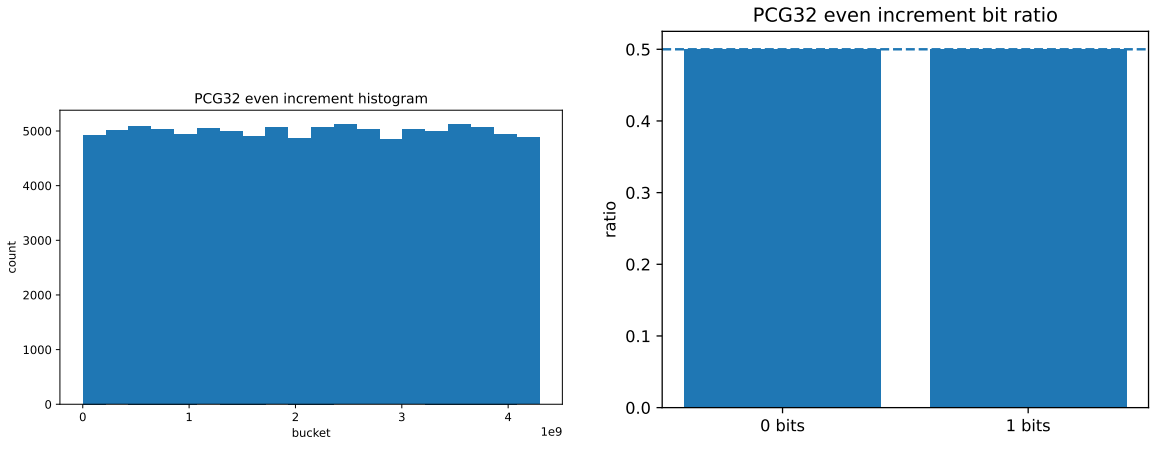

Figure 37: Histogram and bit ratio for the even-increment PCG32 generator

Figure 36 shows that consecutive outputs cover the square of possible values without collapsing into a trivial pattern.

Figure 37 demonstrates that a generator with a broken period condition may still look acceptable in simple empirical tests. This is why theoretical conditions are necessary in addition to plots.

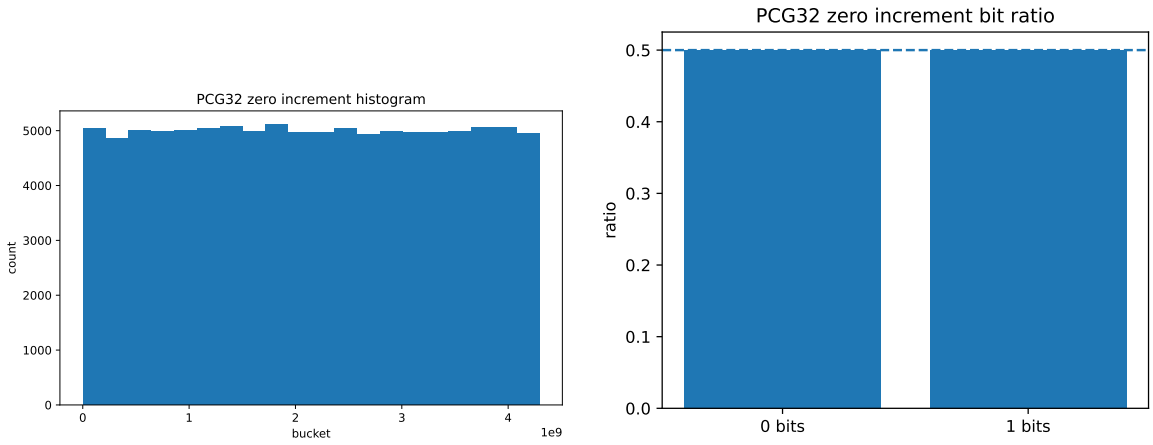

Figure 38: Histogram and bit ratio for the zero-increment PCG32 generator

Figure 38 shows the variant with zero increment. The short-range statistics may still look surprisingly normal, but the recurrence no longer has the full PCG cycle structure. This is another example where empirical plots alone can be too weak as a correctness criterion.

Overall, the experiments show that modern generators can hide many defects in short one-dimensional tests. Xorshift exposes the zero-state problem immediately, while PCG can appear visually reasonable even with broken period conditions. Therefore, both implementation details and theoretical parameter constraints remain essential.

6 Conclusion and Comparison of PRNG Methods

This section summarizes and compares the pseudorandom number generators discussed in the previous chapters. The goal is to highlight their main differences in terms of period length, computational efficiency, statistical quality, and structural complexity.

6.1 General Comparison Criteria

Pseudorandom number generators can be evaluated according to several fundamental criteria:

- **Period length:** the length of the cycle before the sequence repeats.
- **Computational efficiency:** the cost of generating a single value.
- **Statistical quality:** how well the output approximates a truly random sequence.
- **Structural complexity:** mathematical and implementation complexity of the generator.

These criteria are often in conflict; for example, higher statistical quality typically comes at the cost of increased complexity. This is why practical evaluation usually combines mathematical period analysis, implementation-level considerations and empirical tests such as histogram checks, bit-level statistics and more advanced test suites [1, 2].

6.2 Classical Generators

Classical methods such as linear congruential generators (LCG) and Lehmer generators are based on simple modular arithmetic:

$$x_{n+1} = (ax_n + c) \bmod m$$

or in the multiplicative case:

$$x_{n+1} = ax_n \bmod m.$$

These generators are computationally very efficient and easy to implement. However, their statistical properties depend strongly on parameter choice. The period of an LCG is at most m, and full period is achieved only under the Hull–Dobell conditions [3]. A known drawback is their tendency to exhibit geometric structure when viewed in higher dimensions, which reduces their suitability for high-quality simulations [5].

6.3 Recursive and Shift-Based Generators

More advanced recursive methods such as lagged Fibonacci generators (LFG) and linear feedback shift registers (LFSR) improve statistical properties by introducing longer memory into the state:

- LFG uses previous values in a recurrence relation.
- LFSR operates on bit-level linear recurrences over finite fields.

These methods generally achieve longer periods and better distribution properties than simple LCGs, while maintaining moderate computational cost. However, as shown in the experiments, poor taps or weak initialization can still create short cycles or visible structure. This makes parameter selection essential, especially for linear generators over finite fields [10].

6.4 Modern High-Quality Generators

Modern generators such as the Mersenne Twister, PCG, and xorshift-based methods are designed to optimize statistical quality and period length.

The Mersenne Twister achieves an extremely large period of $2^{19937} - 1$, making repetition practically impossible in most ordinary applications [6]. PCG improves statistical behavior by applying permutation functions to linear congruential states, while maintaining efficiency [9]. Xorshift generators rely on bitwise XOR and shift operations, providing extremely fast generation with good statistical performance when properly parameterized [8].

6.5 Trade-offs Between Methods

There is no single optimal PRNG for all applications. Instead, different methods optimize different aspects:

- **LCG / Lehmer:** very fast, but weaker statistical quality and strong dependence on parameters.
- **LFSR / LFG:** better structure and longer memory, but sensitive to taps and initialization.
- **Mersenne Twister:** excellent general-purpose statistical properties and very long period, but larger state complexity.
- **PCG / Xorshift:** modern balance between speed, small implementation size and practical statistical quality.

6.6 Final Remarks

Pseudorandom number generation is fundamentally a trade-off between simplicity, speed, and statistical quality. While classical methods such as LCG provide insight into the structure of deterministic sequences, modern generators achieve practical randomness suitable for large-scale simulations and computational applications.

The experiments in this work also show that visual plots and simple statistics are useful, but not sufficient on their own. A generator may look acceptable in a histogram while still having a poor period, bad bit-level behavior or structural correlations. Therefore, understanding the underlying mathematical structure is essential for selecting appropriate methods in scientific computing, cryptography-related contexts and numerical simulations.

References

[1] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. 3rd ed. Addison-Wesley, 1997.

[2] Pierre L'Ecuyer and Richard Simard. „TestU01: A C Library for Empirical Testing of Random Number Generators". In: *ACM Transactions on Mathematical Software* 33.4 (2007). DOI: 10.1145/1268776.1268777.

[3] T. E. Hull and A. R. Dobell. „Random Number Generators". In: *SIAM Review* 4.3 (1962), pp. 230–254. DOI: 10.1137/1004061.

[4] D. H. Lehmer. „Mathematical Methods in Large-Scale Computing Units". In: *Proceedings of the Second Symposium on Large-Scale Digital Calculating Machinery*. Harvard University Press, 1951, pp. 141–146.

[5] Pierre L'Ecuyer. „Tables of Linear Congruential Generators of Different Sizes and Good Lattice Structure". In: *Mathematics of Computation* 68.225 (1999), pp. 249–260. DOI: 10.1090/S0025-5718-99-00996-5.

[6] Makoto Matsumoto and Takuji Nishimura. „Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator". In: *ACM Transactions on Modeling and Computer Simulation* 8.1 (1998), pp. 3–30. DOI: 10.1145/272991.272995.

[7] Mutsuo Saito and Makoto Matsumoto. „Variants of Mersenne Twister Suitable for Graphic Processors". In: *ACM Transactions on Mathematical Software* 39.2 (2013). DOI: 10.1145/2427023.2427028.

[8] George Marsaglia. „Xorshift RNGs". In: *Journal of Statistical Software* 8.14 (2003), pp. 1–6. DOI: 10.18637/jss.v008.i14.

[9] Melissa E. O'Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. Tech. rep. HMC-CS-2014-0905. Harvey Mudd College, 2014.

[10] Solomon W. Golomb. *Shift Register Sequences*. Holden-Day, 1967.