Investigating Xoroshiro128++ PRNG in Procedural Generation of Minecraft Chunk Structures.

Vincent Rionarlie - 13524031
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: vinctlie06@gmail.com , 13524031@std.stei.itb.ac.id

Abstract*—As a world-based adventure and survival-themed game, Minecraft grants its players the freedom to create and destroy in any worlds as they will. An important element of said freedom is the randomized topographical generation of each world, unique to a certain—almost infinitesimal—degree. The astonishing terrain generation algorithm in Minecraft involves the implementation of pseudorandom number generators (PRNGs), which one of them is Xoroshiro128++. This study is aimed to investigate the correlation between Xoroshiro128++ and the mathematical concepts of Boolean algebra and modular arithmetic, as well as computer's binary representation. Aside of that, this study also aims to test the randomness of the said PRNG.

***Keywords*—Xoroshiro128++, PRNG, Minecraft, Boolean, Number Theory, Random**

I. Introduction

The invention of computer technologies, followed by the internet, has led to the rise of a new, very well-known branch of entertainment for people to choose amongst, such as video games. Since its early emergence in the 1990s, there has currently been around 5 million video games worldwide. One of which, as of October 2023, has claimed the Guinness World Record for the "Best-Selling Video Game of All Time" by having sold over 300 million copies. The game is *Minecraft*, a survival-based action-adventure game, developed by Mojang Studios and now owned by Microsoft.

As a 3D block-based sandbox game, *Minecraft* offers indefinite freedom for their players. Players control a blocky character that are allowed to do anything, including destroying and crafting anything as they will. Players' travelling distance isn't limited, meaning they could keep going beyond whatever is viewed 'too far'. Above all, there are quintillions unique worlds within the game. All of this is made possible through the randomization in *Minecraft* terrain generation process.

Unlike a lot of other games whose assets are crafted manually, *Minecraft* uses a process called *Procedural Content Generation* (PCG) to construct each world. Within the process, a number of *pseudo-random number generators* (PRNGs) are used to determine how the world's overall structure—upper ground terrain, underground caving layouts, building structures, biome, etc.—is generated. Any kind of PRNG implements general discrete mathematical concepts ranges from Boolean algebra to the classic modular arithmetic in number theory. This paper aims to investigate the implementation of those concepts in a specific kind of PRNG—*Xoroshiro128++*, as well as exploring the overall effects throughout *Minecraft* world generation.

II. Minecraft

As its name suggests, *Minecraft* is a game focusing primarily on the actions of mining/destroying and crafting/building. It appears as a 3D block-themed adventure game, where every element throughout the game has the shape of blocks—or at least constructed from blocky pixels—retaining its distinct blocky appearance. The game has 5 different game modes, starting with survival mode as the default, and the rest—hardcore, creative, adventure, and spectator—serves as additional, but still vital modes for players to experiment with. Within the game, players control a character each in a 3D surrounding world, as well as being given indefinite freedom, as they are allowed to do anything, including chopping trees, mining ores, crafting tools, building houses, fighting monsters, travelling the boundless open world, leaving the rest to the players' imagination.

Before starting gameplay, players will need to setup their world, including setting up a pre-existing seed for their world, or let the game randomize for them. World seed contains a length of 64 bits or a maximum decimal number of about 18.4 $\times 10^{18}$, which is then passed down as parameter in almost all elements' procedural generation process. Some of the processes further modifies the seed to match the required value type necessary for each process. As mentioned, the seed's length indicates the number of 2^{64} or about 18.4 quintillion possible unique worlds generated.

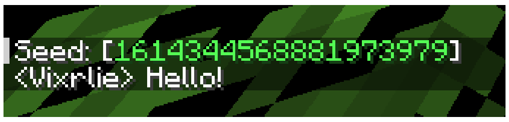
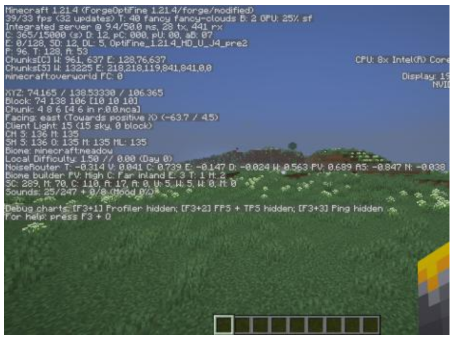

Seed: [1614344568881973979]
<Vixrlie> Hello!

Fig. 1. Example of World Seed

Minecraft is a block-based game, which means players destroy and craft things out of blocks. The player themself is made up of the dimension of 1 block unit lengthwise and widthwise, and 2 blocks vertically. To ease up calculations, *Minecraft* consists of a coordinate system, which the players can also access. Being a 3D game, the coordinate system consists of X, Y, and Z axes, with positive value pointing to east, south, and upward respectively. Coordinate value measures a block's lowest northwest corner, i.e. a block's coordinate of (0.0, 1.0, 0.0) refers to its lower northwest corner. Unlike blocks, which coordinate can only be whole number (or halved for certain blocks), players coordinate values—obviously—have big floating points as a consequence of smooth movement in the game.



```
Minecraft 1.21.4 (ForgeOptiFine 1.21.4/forge/modified)
39/33 fps (32 updates) T: 40 fancy fancy-clouds B: 2 GPU: 25% sf
Integrated server @ 9.4/50.0 ms, 28 tx, 441 rx
C: 365/15000 (s) D: 12, pC: 000, pU: 00, aB: 07
E: 0/128, SD: 12, DL: 5, OptiFine_1.21.4_HD_U_J4_pre2
P: 96, T: 128, A: 53
Chunks[C] W: 961, 637 E: 128,76,637
Chunks[S] W: 13225 E: 218,218,119,841,841,0,0
minecraft:overworld FC: 0

XYZ: 74.165 / 138.53330 / 106.365
Block: 74 138 106 [10 10 10]
Chunk: 4 8 6 [4 6 in r.0.0.mca]
Facing: east (Towards positive X) (-63.7 / 45)
Client Light: 15 (15 sky, 0 block)
CH S: 136 M: 135
SH S: 136 O: 135 M: 135 ML: 135
Biome: minecraft:meadow
Local Difficulty: 1.50 // 0.00 (Day 0)
NoiseRouter T: -0.314 V: 0.041 C: 0.739 E: -0.147 D: -0.024 W: 0.563 PV: 0.689 AS: -0.847 N: -0.038
Biome builder PV: High C: Far inland E: 3 T: 1 H: 2
SC: 289, M: 70, C: 110, A: 17, A: 0, U: 5, W: 5, W: 0, M: 0
Sounds: 25/247 + 0/8 (Mood 0%)

Debug charts: [F3+1] Profiler hidden; [F3+2] FPS + TPS hidden; [F3+3] Ping hidden
For help: press F3 + Q
```

```
CPU: 8x Intel(R) Core
Display: 19
NVID
```

Fig. 2. Coordinate System

The vast world of *Minecraft* itself is made up of millions of the smallest block grouping units, called chunks. A chunk has the dimension of 16×16 blocks in X and Z axes. Any two points in the Y axis belong in the same chunk as long as their X and Z axes are within the chunk itself. In *Minecraft* procedural generation, a world's raw terrain (surface topology and caving) is generated per chunk, which works by combining multiple octaves of Perlin noise—an algorithm which creates smooth monochromic square image—used to simulate terrain shapes.

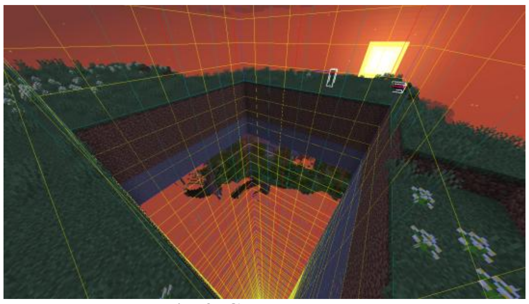


Fig. 3. Chunk Borders

III. Procedural Content Generation

In animation and game development, a lot of assets are required, including base terrain, environmental decorations, building structures, and other theme-specific elements. Content and asset generation can be done manually, where designers sketch and render the assets by hand. This method is a common practice for small to medium scale productions within animation and game development, with the primary reason being the scale and size efficiency, while also preserving the core ideas of the designer. The manual method is no longer relevant as the scale grows larger, except for some major production studios.

For large scale productions, especially in open-world settings, it is arguably more efficient for studios to use an automated generation method, with the cost of losing authenticity over the contents. This is where *procedural content generation* (PCG) plays a critical role.

In PCG, instead of manually crafting generation ideas out of nothing, designers code an algorithm to automate the assets generation. Once executed, the code will create a pseudo-random structure, whose constraints are determined by the designers themselves. In animation developments, procedural generation is often used to create proper assets after assets to use. In game creations, the method is used more directly. Game developers utilize PCG to create assets during or before runtime, whose players can interact with in real time.

In practice, PCG often uses *pseudo-random number generators* (PRNGs) to create unpredictability during assets creation. The pseudo-random number—or commonly called seed—is later passed on as a parameter for generation functions, i.e. graph grammar-based function for dungeon-based games, noise functions and L-system for 3D infinite world game, etc.

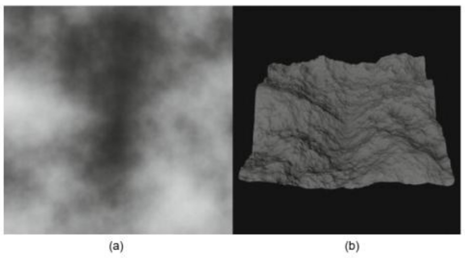


Fig. 4. (a) Example of Perlin Noise, (b) Generated Terrain [6]

IV. Boolean Algebra

A. General Definition

Boolean algebra is a branch of both discrete mathematics that explores the calculations of Boolean values—variable whose values varies within a set of two elements, such as 0 for false, and 1 for true. The theory was first proposed and created by George Boole in his book "The Mathematical Analysis of Logic", published in 1847. It was further refined in his latter book "The Laws of Thought" where he proposed the basic principles of logic. The principles are then formulated under the work of Marshall Stone and Alfred Tarski in 1930s into the new mathematical discipline: Boolean algebra.

B. Boolean Algebra Laws

Boolean algebra is defined as a Boolean ring consisting of the universe U—a set of integers, three operators for the universe, such as two binary operators of addition and multiplication (+ and ·), and a unary complement operator ('), as well as the integers 0 and 1. A Boolean ring follows a set of laws in Table. 1

Table. 1. Boolean Algebra Laws

Identity Law (1) $a + 0 = a,$ (2) $a \cdot 1 = \text{a},$	Involution Law (13) $(a')' = a$
Commutative Law (3) $a + b = b + a,$ (4) $a \cdot b = b \cdot a,$	Absorption Law (14) $a + (ab) = a,$ (15) $a(a + b) = a,$
Distributive Law (5) $a \cdot (b + c) = (a \cdot b) + (a \cdot c),$ (6) $a + (b \cdot c) = (a + b) \cdot (a \cdot c),$	Associative Law (16) $a + (b + c) = (a + b) + c,$ (17) $a \cdot (bc) = (ab) \cdot c,$
Idempotent Law (7) $a + a = a,$	De Morgan's Law (18) $(a + b)' = a'b'$

(8) $a \cdot a = a,$	*(19)* $(ab)' = a' + b',$
Complement Law (9) $a + a' = 1,$ (10) $a \cdot a' = 0,$	De Morgan's Law (20) $0' = 1$ (21) $1' = 0$
Dominant Law (11) $a \cdot 0 = 0,$ (12) $a + 1 = 1,$	

The laws share similarities with the laws underlying the set theory. As such, they are structurally equivalent to each other. For instance, addition and multiplication operator in Boolean algebra represents the union and intersection operator in set theory respectively.

C. Boolean Function

In order to obtain more varying calculation outputs, there exists Boolean function whose arguments and the function itself hold a value each within the two-element set of $\{0,1\}$. More formally, let us denote $B = \{0,1\}$. B^n consists of all sets of its member sequences from set B (binary vectors) to the length of n, so that:

$$B^n = \{ (b_1, b_2, \dots, b_n) \mid b_i \in B \} \qquad (1)$$

For any Boolean function f of n variables, it is defined as a mapping of B^n such that:

$$f: B^n \to B \qquad (2)$$

Provided a binary vector $b = (b_1, b_2, \dots, b_n)$, b is called a *true point* if $f(b) = 1$ and a *false point* if $f(b) = 0$. A Boolean function f is made up of some terms—product of literals. A literal is any Boolean variable b_i as well as its complement. A function's degree is the amount of its literal. Following this definition, a Boolean function can take canonical form in *Minterm (3)*—or *Sum of Products* (SOP), where the function minimizes the number of terms by adding all possible products of literals—and *Maxterm (4)*—or *Product of Sums* (POS) as opposed to Minterm, by multiplying all possible sum of literals. For example:

$$f(x,y) = xy' + x'y \qquad (3)$$
$$f(x,y) = (x + y')(x' + y) \qquad (4)$$

D. Logic Gates

By applying Boolean functions, we can obtain the implementation of logic gates by combining different sets of terms of the second degree, as proved by Shannon in 1938. Logic gate is defined as a digital gate that allows data to travel through, where data is defined as a stream of *true* / 1 value. By writing down all possible combinations of function return value for two Boolean variables, we get to see some very commonly used logic gates that are used widely in the field of electrical and computer engineering. Consider two variables A and B, as well as functions $F_0 \dots F_{n-1}$, where $n = 2^{2^2} = 16$, we will get all of the possible combinations of truth tables for A and B.

Table. 2. Truth Table Combinations

A	*B*	*F_0*	*F_1*	*F_2*	*F_3*	*F_4*	*F_5*	*F_6*	*F_7*
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

A	*B*	*F_8*	*F_9*	*F_{10}*	*F_{11}*	*F_{12}*	*F_{13}*	*F_{14}*	*F_{15}*
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Some of the functions return consistent values to the commonly used logic operators. The list can be seen below.

Table. 3. Common Logic Gates List

Index	Symbol	Name
F_0	0	FALSE
F_1	*$A \cdot B$*	AND
F_6	*$A \oplus B$*	XOR
F_7	*$A+B$*	OR
F_8	*$(A+B)'$*	NOR

Index	Symbol	Name
F_9	*$A \odot B$*	XNOR
F_{10}	*B'*	NOT (B)
F_{12}	*A'*	NOT (A)
F_{14}	*$(A \cdot B)'$*	NAND
F_{15}	1	TRUE

V. Modular Arithmetics

A. General Definition

In mathematics, the discipline that studies numbers is called the *Number Theory*. It explains how numbers interact with each other, as well as the properties specific numbers have. It also, certainly, explores the division properties of numbers, which then gave rise to *Modular Arithmetic*. It is another branch of discrete mathematical field that studies specifically the division and remainder properties of integers.

B. Division and Remainder Rules

For any $a, b, c \in \mathbb{Z}$ where $a \neq 0$, then $a \mid b \leftarrow b = ac$, which reads a divides b if there exists an integer c such that $b = ac$. The definition is then extended with the *Euclidean Division Lemma*, that states for any $a, b \in \mathbb{Z}$ where $b > 0$, then $a = bq + r$ for any $q, r \in \mathbb{Z}$ and $0 \leq r < n$.

Another important concept is the *Greatest Common Divisor* (GCD). GCD of two numbers is equal to the largest positive integer that divides both of them. By notation:

$$GCD(a,b) = c \leftarrow \{ c \in \mathbb{N}, (c|a), (c|b) \} \qquad (1)$$
$$0 < c \leq \max(a,b) \qquad (2)$$

The search for an integer c is not intuitive for arbitrarily large a and b. One can implement the *Euclidean Algorithm* to find the GCD of large a and b. Consider $a \geq b$ and $r_0 = a$ and $r_1 = b$:

$$r_0 = r_1 q_1 + r_2 \quad , 0 \leq r_2 < r_1 \qquad (3)$$
$$r_1 = r_2 q_2 + r_3 \quad , 0 \leq r_3 < r_2 \qquad (4)$$
$$\vdots$$
$$r_{n-1} = r_n q_n + 0 \qquad (5)$$

$$GCD(a,b) = GCD(r_0, r_1) = GCD(r_1, r_2) = GCD(r_n, 0) = r_n = c \qquad (6)$$

C. Modulo Operation

Modulo operation outputs a division remainder between two integers. This concept is strongly tied to the division rules, as it is defined based on them. For any $a, m, r \in \mathbb{Z}$, $a \bmod m = r \leftrightarrow$

$a = mq + r$ where $0 \leq r < m$. The set of all possible values of r is defined as $\{0,1,\dots,m\}$.

Consequently, there are infinite possibilities of value a that shares the same result when it is done the modulo operations with m. Therefore, a congruence ($\equiv$) is used to indicate a similarity in modulo result between two values a and b, such that $a \equiv b \bmod m$, where it requires $m \mid (a - b)$.

VI. Binary Logic

A. *Binary Number Representation*

Computer machines represent and scan information in a base-2 numbering system, often called *binary* notation. Unlike our daily base-10 or decimal notation, binary notation represents numbers with only two digits: 0 and 1. It resets to zero every 2 numbers, for example 1_{10} becomes 1_2 , 2_{10} becomes 10_2 , 3_{10} becomes 11_2, and 4_{10} becomes 100_2. The leftmost digit of the binary number is assigned the index of 0, with each following digits to the right having an index incremented by one. Each digit in a binary number is called a *bit*. A conversion from binary system to decimal system can be done by calculating the sum of each binary digit multiplied by 2 to the power of the digit's index. For instance, $10100_2 = 0{\cdot}2^0 + 0{\cdot}2^1 + 1{\cdot}2^2 + 0{\cdot}2^3 + 1{\cdot}2^4 = 20_{10}$. Oppositely, a conversion of decimal number to binary number is done by repeatedly: dividing a decimal number by two, leaving the floor result for next division, and writing down the remainder with the first division remainder being the leftmost digit of the binary number. As an example of 10_{10}, $10{\div}2 = 5{+}0 \rightarrow 5{\div}2 = 2{+}1 \rightarrow 2{\div}2 = 1{+}0 \rightarrow 1{\div}2 = 0{+}1$, which results to 1010_2.

B. *Primitive Data Types*

One of the basic implementations of binary notation in computer machine is within the primitive data types present in programming languages. First of all, binary numbers are generally stored in *bits* or a collection of 8 *bits*, called a *byte*. Every primitive data type is assigned its own size.

Table. 4. Primitive Data Types in Java Language

Name	Size (*bits*)	Value Stored
Boolean	1	True/False (0 or 1)
Byte	8	Byte (-128 to 127)
Char	16	Character representation in ASCII (0 to 255)
Short	16	Integer (-31,768 to 31,767)
Int	32	Integer (-2,147,483,648 to 2,147,483,647)
Long	64	Integer (-2^{63} to 2^{63}-1)
Float	32	Real Number (up to 7 decimal points)
Double	64	Real Number (up to 16 decimal points)

For integer type primitives, negative values are obtained with the implementation of *two's-complement method*. In this method, the rightmost *bit* is regarded as a *sign bit*, where 0 indicates positive and 1 indicates negative. The method regards the *sign bit* as -2^{n-1} instead of 2^{n-1}, which, when it gets multiplied by 1, will result in the smallest negative number possible for the data type. The addition of 1 to any other bits consistently signifies positive addition to both positive and negative number.

C. *Bitwise Operator*

Similar to decimal numbers, binary numbers can be calculated with basic operations, including addition, subtraction, multiplication, division, and even modulo. These operators work just like decimal operators, just a bit more unintuitive. For example:

$$\begin{array}{r} 10010_2 \\ \underline{10100_2 +} \\ 100110_2 \end{array} \qquad \begin{array}{r} 10101_2 \\ \underline{1001_2 -} \\ 1100_2 \end{array} \qquad \begin{array}{r} 10_2 \\ \underline{11_2 \times} \\ 10 \\ \underline{10:} \\ 110_2 \end{array}$$

Division and modulo operator work similarly as well, but will not be displayed due to simplicity.

Aside from the basic operators, there are some other crucial binary operators in computer organization, called the *bitwise operators*. They share a similarity with the logic gate operators showed in **Table. 3** The key difference is that the operators check every *bit* of both of the binary values given at the same time, as opposed to Boolean operators that only check two *bits* values. Some *bitwise* operators are shown below.

Table. 5. Bitwise Operators in C Language

Operators	Name	Usage	Description
&	Bitwise AND	a & b	AND operator on every *a* and *b*'s bits with the same indices
|	Bitwise OR	a | b	OR operator on every *a* and *b*'s bits with the same indices
^	Bitwise XOR	a ^ b	XOR operator on every *a* and *b*'s bits with the same indices
~	Bitwise NOT	~a	Negates every bit of *a*
<<	Left Shift	a << b	Shift every *a*'s bit to the left by *b* times and fill the rightmost with 0s
>>	Right Shift	a >> b	Shift every *a*'s bit to the right by *b* times, and fill the leftmost with data type's preference

VII. Investigation Strategy

A. *Minecraft Commercial Usage License*

Minecraft is a close-source software. It's stated in the Microsoft Service Agreement section 8b point (ii) under Software License, that, user does not have the license to "disassemble, decompile, decrypt, hack, emulate, exploit, or reverse engineer any software or other aspect of the Services that is included in or accessible through the Services, except and only to the extent that the applicable copyright law expressly permits doing so". The paper's author fully respects the rule. For research purpose, the author utilized a framework that allows for *Minecraft* modification legally. As such, the author examined the local game source code to gain better understanding on how the specific PRNG—Xoroshiro128++—works. The author will not redistribute any important assets of the game, including the source code. Hence, to provide an informative overview of the algorithm, the author will only uncover the main logic of the algorithm, as well as creating a custom script to illustrate how the algorithm functions.

B. *Custom Code Construction*

After inspecting the general overview of *Minecraft*'s source code, the author implements a *Xoroshiro128++* custom program in Java language. To reduce resemblance with *Minecraft*'s source code, the author constructs the code by investigating the original *Xoroshiro128++* work done by David Blackman and Sebastiano Vigna [13].

C. *Randomness Test*

A good RNG is distinguished by its low chance to produce a repeated sequence of numbers. To test whether *Xoroshiro128++* is a good pseudo-RNG, the author utilized the tool made by the US *National Institute of Standards and Technology* (NIST), namely Nist Test Suite SP 800-22. It consists of 15 different tests, which each produces a certain probability value. The author will only focus on the probability values and calculate its overall value to determine if it is considered a good PRNG.

VIII. Xoroshiro128++ Implementation

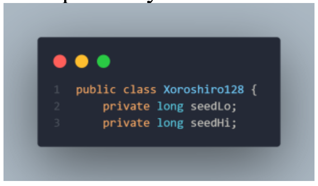
A. *Mathematical and Computational Concepts*

Xoroshiro128++ acts as a PRNG that applies linear transformation to update two groups of *words* of a larger state array. It is done to obtain a good parallelizability inside large scale CPUs. It works by combining a XOR, rotation, shift, and again a rotation—hence its name—onto a *word* of 64 bits (a randomized or preset seed).

In the program's implementation, *Xoroshiro128++* relies heavily on the three main operators above. At first, a *Xoroshiro128++* function is passed two arguments of *long* data type seeds, namely low seed and high seed. At initialization, the function is given an arbitrary value that'll determine the seeds number. The value was then calculated with the three main operators to achieve the *result* wanted. The calculation consists of left rotation, which operates a left shift and wraps the leftmost shifted values to the rightmost bits. Other than that, the calculation combines a set of bitwise XOR function between the low and high seed, as well as left shifts that is equivalent with multiplying by 2^n for every $x \ll n$. The values passed into the operators functions are of precise choices that preserves the distribution and randomness of the PRNG.

B. *Sample Algorithm*

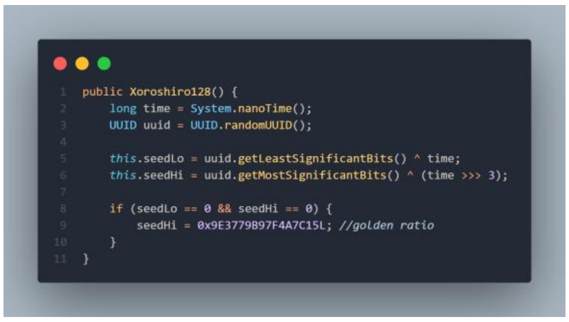
Firstly, the author defined a class *Xoroshiro128* with two *long* type attributes, *seedLo* and *seedHi*, each representing low seed and high seed respectively.



```java
public class Xoroshiro128 {
    private long seedLo;
    private long seedHi;
```

Fig. 5. *Xoroshiro128++* Class

When initialized as a *new* class, *Xoroshiro128* will undergo a function that sets its initial values to pseudorandom values-alike, which is further calculated based on computer's system elapsed runtime—at the moment it is called—in nanoseconds, as well as the significant bits of a UUID (Universally Unique Identifier) —an identifier whose value created through the process of another PRNG, usually used for file naming and session tokens. It also implements the randomness of the *golden ratio* binary value in the data type *long*, in rare cases where the variables *seedLo* and *seedHi* are both 0s.



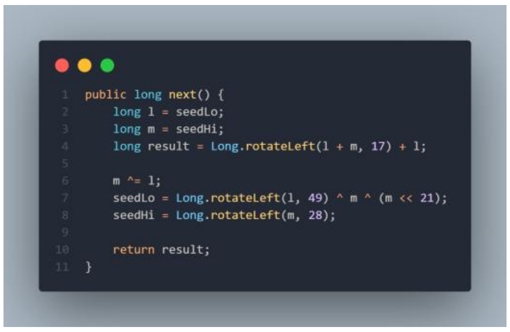
```java
public Xoroshiro128() {
    long time = System.nanoTime();
    UUID uuid = UUID.randomUUID();

    this.seedLo = uuid.getLeastSignificantBits() ^ time;
    this.seedHi = uuid.getMostSignificantBits() ^ (time >>> 3);

    if (seedLo == 0 && seedHi == 0) {
        seedHi = 0x9E3779B97F4A7C15L; //golden ratio
    }
}
```

Fig. 6. *Xoroshiro128++* Class Initialization

The core of *Xoroshiro128++* comes from its *next* function, where it generates the pseudo-random number used for other utilities, including *Minecraft* random chunk generator. It first calculates *result* as the function return value, which is done through a combination of binary rotation and addition operators. Following that, the function further modifies the class's own value. It's done through sets of XOR, rotation, and left shift operators.



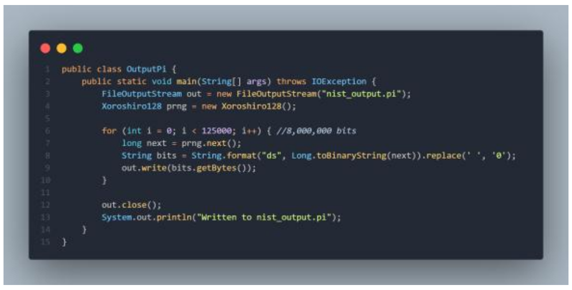
```java
public long next() {
    long l = seedLo;
    long m = seedHi;
    long result = Long.rotateLeft(l + m, 17) + l;

    m ^= l;
    seedLo = Long.rotateLeft(l, 49) ^ m ^ (m << 21);
    seedHi = Long.rotateLeft(m, 28);

    return result;
}
```

Fig. 7. *Xoroshiro128++* Randomizer Function

For testing purpose, a main function is made inside the class *OutputPi* to implements the *Xoroshiro128* class and functions. The main function prints out 125,000 values generated by *Xoroshiro128*'s *next* function as binary strings of 64 bits each. It is then written into a file named *nist_output.pi*.



```java
public class OutputPi {
    public static void main(String[] args) throws IOException {
        FileOutputStream out = new FileOutputStream("nist_output.pi");
        Xoroshiro128 prng = new Xoroshiro128();

        for (int i = 0; i < 125000; i++) { //8,000,000 bits
            long next = prng.next();
            String bits = String.format("ds", Long.toBinaryString(next)).replace(' ', '0');
            out.write(bits.getBytes());
        }

        out.close();
        System.out.println("Written to nist_output.pi");
    }
}
```

Fig. 8. Main Output Function

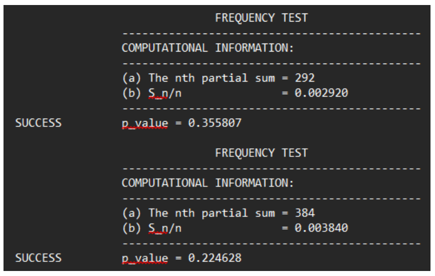
C. *NIST Test Suite*

The implemented *Xoroshiro128* code wrote binary strings into the file *nist_output.pi*. This file doesn't necessarily need to be under the file extension of .pi, but doing so provides convenience as it's most compatible with the *NIST Test Suite* used. The *Xoroshiro128* program generated a sequence of binary strings with the length of 64 each multiplied by 125,000, which results in a sequence of length 8,000,000. The sequence is then divided into bitstream segments of length 100,000 to be tested. It indicates that the test is conducted 80 times.

The test generates a value called *P-value*, which is defined as the probability (0-1) that a *perfect* RNG would have produced a sequence less random than the RNG/PRNG being tested. Thus, a *P-value* of 1 indicates that the RNG tested would 100% produce a perfect random sequence. The tool provides 15 different tests, with different parameters that contribute to each test *P-value.*

- Frequency (Monobit) Test
- Frequency Test within a Block
- The Runs Test
- Tests for the Longest-Run-of-Ones in a Block
- The Binary Matrix Rank Test
- The Discrete Fourier Transform (Spectral) Test
- The Non-overlapping Template Matching Test
- The Overlapping Template Matching Test
- Maurer's "Universal Statistical" Test
- The Serial Test
- The Approximate Entropy Test
- The Cumulative Sums (Cusums) Test
- The Random Excursions Test
- The Random Excursions Variant Test

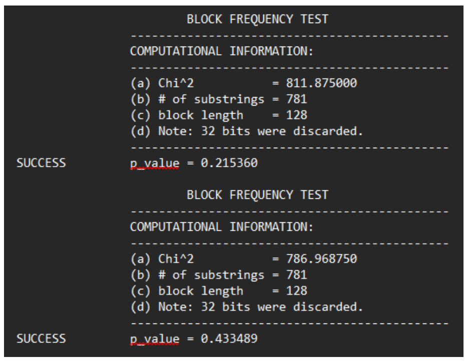
Out of all 15 different tests, author chose to focus on *monobit frequency test*, *frequency test within block*, and *approximate entropy test*, but not limited to them. The former test focuses on the proportion of 0s and 1s among the test subject, which is assessed as the closeness to 50% 0s and 50% 1s as expected from a truly random number and converted into *P-value*. *Frequency within a block* test aims to detect localized deviations from the ideal 50% frequency of 1s. It is done by decomposing the test sequence into a number of nonoverlapping subsequences, and then applying a chi square test. Finally, the latter entropy test focuses on the frequency of all possible overlapping blocks of two consecutive/adjacent lengths (*m* and *m+1*) against the expected result for a random sequence, which is also then converted into *P-value*. For reference, only two samples of each test will be displayed.



```
                        FREQUENCY TEST
              ---------------------------------------------
              COMPUTATIONAL INFORMATION:
              ---------------------------------------------
              (a) The nth partial sum = 292
              (b) S_n/n               = 0.002920
              ---------------------------------------------
SUCCESS       p_value = 0.355807

                        FREQUENCY TEST
              ---------------------------------------------
              COMPUTATIONAL INFORMATION:
              ---------------------------------------------
              (a) The nth partial sum = 384
              (b) S_n/n               = 0.003840
              ---------------------------------------------
SUCCESS       p_value = 0.224628
```

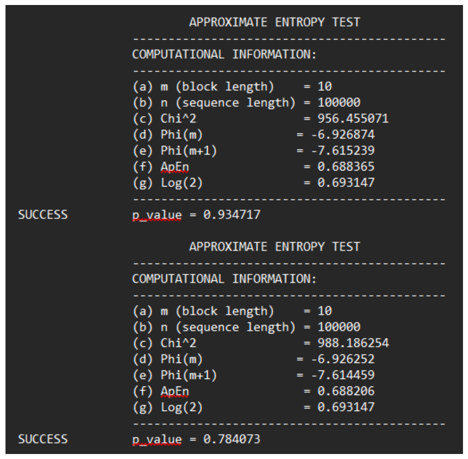
Fig. 9. Frequency Test Sample Results



```
                        BLOCK FREQUENCY TEST
              ---------------------------------------------
              COMPUTATIONAL INFORMATION:
              ---------------------------------------------
              (a) Chi^2           = 811.875000
              (b) # of substrings = 781
              (c) block length    = 128
              (d) Note: 32 bits were discarded.
              ---------------------------------------------
SUCCESS       p_value = 0.215360

                        BLOCK FREQUENCY TEST
              ---------------------------------------------
              COMPUTATIONAL INFORMATION:
              ---------------------------------------------
              (a) Chi^2           = 786.968750
              (b) # of substrings = 781
              (c) block length    = 128
              (d) Note: 32 bits were discarded.
              ---------------------------------------------
SUCCESS       p_value = 0.433489
```

Fig. 10. Block Frequency Test Sample Results

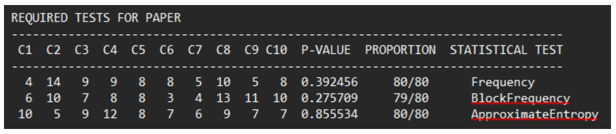


```
                        APPROXIMATE ENTROPY TEST
              ---------------------------------------------
              COMPUTATIONAL INFORMATION:
              ---------------------------------------------
              (a) m (block length)    = 10
              (b) n (sequence length) = 100000
              (c) Chi^2               = 956.455071
              (d) Phi(m)             = -6.926874
              (e) Phi(m+1)           = -7.615239
              (f) ApEn                = 0.688365
              (g) Log(2)              = 0.693147
              ---------------------------------------------
SUCCESS       p_value = 0.934717

                        APPROXIMATE ENTROPY TEST
              ---------------------------------------------
              COMPUTATIONAL INFORMATION:
              ---------------------------------------------
              (a) m (block length)    = 10
              (b) n (sequence length) = 100000
              (c) Chi^2               = 988.186254
              (d) Phi(m)             = -6.926252
              (e) Phi(m+1)           = -7.614459
              (f) ApEn                = 0.688206
              (g) Log(2)              = 0.693147
              ---------------------------------------------
SUCCESS       p_value = 0.784073
```

Fig. 11. Approximate Entropy Test Sample Results

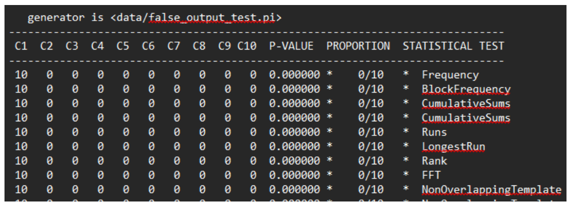
The samples are naturally not representative, as the data size is truly large (8 million bits). Even so, these data provide a better insight for how the values are calculated. To get a better visualization of the test, let us look at the average *p-values* for the three tests.



```
REQUIRED TESTS FOR PAPER
------------------------------------------------------------------------------
 C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
  4  14   9   9   8   8   5  10   5   8  0.392456     80/80      Frequency
  6  10   7   8   8   3   4  13  11  10  0.275709     79/80      BlockFrequency
 10   5   9  12   8   7   6   9   7   7  0.855534     80/80      ApproximateEntropy
```

Fig. 12. Main Tests Average *p-values*

The data shows a lower end value to the test sequence. It is not to worry, since a sequence with *p-value* above the default *a* value (0.01 or 1%) is acceptable as a random number with 99.9% confidence. A simple test to set each sequence to 1 immediately obtains the value of 0.000000 for every tests.



```
   generator is <data/false_output_test.pi>
------------------------------------------------------------------------------
C1  C2  C3  C4  C5  C6  C7  C8  C9 C10  P-VALUE  PROPORTION  STATISTICAL TEST
------------------------------------------------------------------------------
10   0   0   0   0   0   0   0   0   0  0.000000 *    0/10   *  Frequency
10   0   0   0   0   0   0   0   0   0  0.000000 *    0/10   *  BlockFrequency
10   0   0   0   0   0   0   0   0   0  0.000000 *    0/10   *  CumulativeSums
10   0   0   0   0   0   0   0   0   0  0.000000 *    0/10   *  CumulativeSums
10   0   0   0   0   0   0   0   0   0  0.000000 *    0/10   *  Runs
10   0   0   0   0   0   0   0   0   0  0.000000 *    0/10   *  LongestRun
10   0   0   0   0   0   0   0   0   0  0.000000 *    0/10   *  Rank
10   0   0   0   0   0   0   0   0   0  0.000000 *    0/10   *  FFT
10   0   0   0   0   0   0   0   0   0  0.000000 *    0/10   *  NonOverlappingTemplate
```

Fig. 13. False Output Result

As further analysis, the average *p-value* for all of the 15 tests is calculated.

$$p_{avg} = 7.3529029 \div 15 \approx 0.49019352 \qquad (1)$$

As the average *p-value* is 490% larger than *a* value of 0.01, the *Xoroshiro128++* proofs to be a good PRNG. A value at just about 50% fits the pseudorandom number nature itself.

D. *Minecraft Implementation*

In *Minecraft*, *Xoroshiro128++* is used mainly in the generation of random seed in case players have not set one, as well as chunk generators. The implementation of a PRNG is needed to create a natural-occurring terrain and structures. The values returned by PRNG determines a lot of terrain generation aspects. One of the major generations being the environments decorator generation, where trees, grass, and other biome-specific elements are formed. Aside from that, the usage of *Xoroshiro128++* is also apparent if the player hasn't set any seeds before creating the world. The game will attempt almost immediately to create a random *Xoroshiro128++* seed by also utilizing player's system's elapsed time.

IX. Conclusion

By this research, it can be concluded that *Xoroshiro128++* algorithm serves as a very important element of *Minecraft*'s world generation. With the help of the algorithm created by Blackman and Vigna, the game is able to produce such a sophisticated, yet still natural structural design. The implementation of mathematical concepts—Boolean algebra, modular arithmetic, and a little bit of linear algebra, as well as the computational concept of bitwise operations, contributes to the making of such a precisely crafted algorithm. Through further experiments done in this paper, it was also deduced with the help of the tool NIST Test Suite SP800-22, that *Xoroshiro128++* algorithm was considered a good pseudorandom number generator (PRNG) by looking at its average *p-value* which lies at the value 0.49019, significantly higher than *a* value of 0.01. For future references, the research into the detailed mechanism of *Xoroshiro128++* needs to be deepened with further understandings of linear algebra.

X. Appendix

- Source Code: https://github.com/Vixrlie/DiscreteMath.git
- Video Youtube: https://youtu.be/-IGOnYkYsMI

XI. Acknowledgement

The author wants to express their deepest gratitude towards Mr. Rinaldi Munir as their lecturer in Discrete Mathematics Class-01. It has been a very insightful experience to be a student of his, for they have learned a lot of knowledge in a very structured curriculum. It was also through the help of his open-sourced discrete mathematic resources that they have been able to easily navigate between materials for study and research purposes. The author would also thank him for this paperwork assignment through the opportunity for them to explore such a bizarre topic to write as their assignment.

References

[1] Guinness World Records, "Best-selling videogame". Accessed: June 14, 2025. [Online]. Available: https://www.guinnessworldrecords.com/world-records/best-selling-video-game

[2] T. X. Short and T. Adams, Procedural Generation in Game Design, 1st ed., Boca Raton, FL: CRC Press, 2017. Accessed: June 14, 2025. [Online]. Available: https://books.google.co.id/books?id=Rj4PEAAAQBAJ

[3] Rune Skovbo Johansen, "Procedural world potentials: The simulation, functional and planning approaches," blog.runevision.com, 2015. Accessed: Jun. 16, 2025. [Online]. Available: https://blog.runevision.com/2015/08/procedural-world-potentials-simulation.html.

[4] R. Munir, "IF2120 Matematika Diskrit," Aljabar Boolean, Bandung Institute of Technology, 2024. Accessed: Jun. 18, 2025. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/matdis.htm

[5] R. Munir, "IF2120 Matematika Diskrit," Teori Bilangan, Bandung Institute of Technology, 2024. Accessed: Jun. 18, 2025. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/matdis.htm

[6] W. Eck and M. Lamers, "Biological content generation: Evolving game terrains through living organisms," in Lecture Notes in Computer Science, vol. 9027, C. Martinho, A. El Rhalibi, M. Yannakakis, and J. Baalsrud Hauge, Eds. Cham, Switzerland: Springer, 2015, pp. 335–344. doi: 10.1007/978-3-319-16498-4_20. Accessed: Jun.18,2025.

[7] Stanford Encyclopedia of Philosophy, "The Mathematics of Boolean Algebra," first published Jul. 5, 2002; substantive revision Jul. 11, 2018. [Online]. Accessed: Jun. 19, 2025. Available: https://plato.stanford.edu/entries/boolalg-math/.

[8] S. Givant and P. R. Halmos, Introduction to Boolean Algebras, Undergraduate Texts in Mathematics. New York, NY: Springer Science+Business Media, 2008. Accessed: Jun. 19, 2025 [Online]. Available: https://books.google.co.id/books?id=ORILyf8sF2sC

[9] Microsoft Corporation, "Section 8: Software License," *Microsoft Services Agreement*. Accessed: Jun. 19, 2025. [Online]. Available: https://www.microsoft.com/en-id/servicesagreement#8_softwareLicense.

[10] Wolfram Research, "Boolean Function," *MathWorld–A Wolfram Web Resource*. Accessed: Jun. 20, 2025. [Online]. Available: https://mathworld.wolfram.com/BooleanFunction.html.

[11] National Institute of Standards and Technology (NIST), *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, Special Publication 800-22, Rev. 1a, Apr. 2010. Accessed: Jun. 20, 2025. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-22r1a.pdf

[12] R. E. Bryant and D. R. O'Hallaron, Computer Systems: A Programmer's Perspective, 3rd ed. Boston: Pearson, 2015. Accessed: Jun. 20, 2025. [Online]. Available: https://www.cs.sfu.ca/~ashriram/Courses/CS295/assets/books/CSAPP_2016.pdf

[13] D. Blackman and S. Vigna, "Scrambled Linear Pseudorandom Number Generators," arXiv:1805.01407 [cs.DS], 2018. Accessed: Jun. 20, 2025. [Online]. Available: https://vigna.di.unimi.it/ftp/papers/ScrambledLinear.pdf

Declaration

Hereby, I declare that the paper is a work of mine, and not a copy nor translation nor plagiarism of others' work.

Bandung, 20 Juni 2025

[illegible]

Vincent Rionarlie / 13524031