Survey on Random Number Generators

Benjamin Planche

Universität Passau, Germany,
planch01@stud.uni-passau.de,
Home page: https://github.com/Aldream/random-number-generator

Abstract. This survey focus on Random Number Generators (RNGs). In this light, it first attempts to unravel the various definitions of randomness and pseudo-randomness, before detailing the different categories of random generators, based on physical phenomena or computations. Emphasis is put on Linear- and Non-Linear Feedback Shift Registers (LFSRs and NLFSRs), and their implementation. Finally, the difficulty of objectively testing RNGs is discussed, and various test suites are described.

Keywords: RNG, randomness, entropy, pseudorandom, LFSR, NLFSR, test suite, Berlekamp-Massey algorithm

1 Introduction

The generation of random numbers is an important operation for various computational tasks, but not all these applications have the same requirements about the *randomness* of the obtained sequences. This paper has thus for purpose to give an overview of the most common requirements and solutions in the vast domain of random number generation.

In Section 2, we first discuss the various definitions behind randomness. Section 3 develops about the two main categories of RNGs and offers implementations of the famous LFSRs and NLFSRs. In section 4, we explain the problems relative to testing such generators, before presenting some famous test batteries and developing about the *Berlekamp-Massey* algorithm and its implementation. Finally, we conclude this paper in Section 5.

2 Randomness

Randomness is defined as the lack of pattern, predictability or determinism in events. It is thus generally difficult to evaluate if a sequence is truly random, or if we are simply ignorant of a hidden underlying pattern.

Among other matters, information theory is thus studying the properties of a random sequence. This notion is usually defined as being a sequence of independent random variables. However, the elaboration of a more formal and mathematical characterization is quite challenging, and has been the subject of many debates and studies during the last century.

2.1 Definition by Von Mises

The first attempt at defining algorithmic randomness was made by the scientist and mathematician *Richard Edler von Mises* (19 April 1883 14 July 1953). Based on the theory of large numbers, he stated that an infinite sequence of symbols (bits actually) can be considered random if it possesses the frequency stability property, i.e. if the appearance frequencies of the symbols aren't biased and goes to the same expected value (cf. the strong law of large numbers), but also if any sub-sequence selected by a "proper method" isn't biased [1].

The sub-sequence criterion is fundamental in this definition. For instance, if the sequence 10101010 is not biased, we obtain the biased sub-sequence 0000 by selecting only the even positions [1].

This first definition was however unsatisfying, because of the failed attempts to mathematize what the proper method of selection was, and also because of a demonstration by *Jean Ville* in 1939, proving that such a definition only yields an empty set [1]. After the development of the information theory in the middle of the twentieth century, new definitions were proposed in the sixties, reaching some kind of consensus.

2.2 Definition by Martin-Löf

A sequence is random for the Swedish statistician *Per Martin-Löf* if it has no "exceptional and effectively verifiable" property, i.e. has no properties which can be verified by a recursive algorithm. This definition, presented in 1966 and based on the measure theory, is considered the most satisfactory notion of algorithmic randomness. It is considered as a *frequency / measure-theoretic* approach to randomness [1] [1].

2.3 Definition by Levin/Chaitin

Ray Solomonoff (25 July 1926 7 December 2009) and *Andre Nikolaevitch Kolmogorov* (25 April 1903 20 October 1987) developed in the sixties an important measure for the field of information theory: *the complexity of Kolmogorov* [1]. This measure is defined as the length of the shortest program (independently of the machine running it) able to generate the evaluated sequence.

Using this work, *Leonid Anatolievich Levin* (born in 1948) and *Gregory Chaitin* (born in 1947) concluded in 1975 that a random finite string can be considered as a string which requires a program at least as long as itself to be computed. Another way to express it is: "*a random sequence must have an incomprehensible informational content*", i.e. it is impossible to make any sense of it, and thus to use a shorter sequence or program to describe it [1]. This is why this approach is considered as a *complexity / compressibility* one.

2.4 Definition by Schnorr

Taking the predictability approach, *Claus-Peter Schnorr* (born in 1943) used the martingales theory in 1971 to build his definition, which is the following: "a random sequence must not be predictable. No effective strategy should lead to an infinite gain if we bet on the symbols of the sequence" [1].

2.5 Pseudo-Randomness

A numeric sequence is considered statistically random if it contains no recognizable patterns or regularities. Compared to the previous definitions, statistical randomness is less strict and actually doesn't imply objective unpredictability. It thus leaves room to the concept of pseudo-randomness.

A pseudo-random sequence exhibits statistical randomness while being generated by an entirely deterministic causal method. Such sequences are largely used in computer science, to simulate random behaviors, when generating truly random values would be too costly. Indeed, because pseudo-random sequences can be algorithmically generated, they can be used in a much simpler and frequent way.

3 Random Number Generators

Mediums to generate random sequences have been used for a long time in various domains, from politics to cryptology. However, they are not all equal, and most of them suffer from bias which can be or not a problem depending on the usages.

3.1 Definition

A *random number generator* (RNG) is a device which can produce a sequence of random numbers, i.e. a sequence without determinist properties and patterns. Such a device can use physical interactions, computations, or a mix of both, to achieve it.

As defined before, since a purely random sequence can't be described or generated by an algorithm (not recursive), computational methods can only create pseudo-random sequences. Such generators are thus named *pseudo-random number generators* (PNRG).

3.2 Categories

Generators based on physical phenomena: Physical interactions, such as dice tossing or coin flipping, have been traditionally used to make random decisions and generate unpredictable sequences.

However, some physical phenomena are only random in appearances. For instance, in the case of the coin flipping, dynamics rules are applied to the trajectory of the coin. The results seem random only because we can't simply measure the variables of the toss to solve the system. But having even some meager hints about the initial conditions can then help deduce the outcome (for example, the face on top before the coin is tossed may have more chances to be the resulting one).

The best phenomena to use as input for random generators are thus the phenomena possessing *quantum mechanical physical randomness*, especially found in quantum mechanics at the atomic or sub-atomic level. Based on this property, various methods have been implemented to generate random sequences: by measuring the nuclear decay with a Geiger counter, by printing 0 or 1 when a photon is reflected or transmitted by a semi-transparent mirror, etc [2, 1].

If these devices are seen as golden solutions, they are globally too costly to be democratized. Thermal phenomena are for instance easier to detect and offer good results. The noise obtained by amplifying the thermal signal from a transistor or from an atmospheric radio receiver are famous examples, though other solutions have also been developed (based on the comparison of pictures from an agitated scene, on the noise from analog-to-digital converter, etc.) [2].

In computer science, operating systems implement various methods based on the unpredictable inputs/outputs and the behavior of the users. In Unix systems for instance, */dev/urandom* and */dev/random* are device files probing analog sources (mouse, keyboard, disk accesses, etc.) to harvest entropy and thus output random bytes; while *CryptGenRandom* for Windows systems gather entropy though CPU counters, environment variables, threads IDs, etc. In both cases however, entropy tends to decrease during inactivity, which could lead to shortages [3].

Pseudo-Random Number Generators: As it has been said, PRNG are generators based on algorithms. It might seem paradoxical to associate randomness with algorithms, which are by definition determinist. But some cleverly implemented processes can generate pseudo-random sequences with periods long enough for their usages. Because pseudo-random number generators are deterministic, their output sequences are totally defined by their initial configuration, called *state*. The key variable of this state is the *seed* (or *random seed*), a number, or vector, which should be kept secret otherwise anyone could predict the sequence generated from this setting [2, 1].

We present below to famous categories of PRNGs, based on shift registers:

LFSR: A *Linear Feedback Shift Register* (LFSR) is a sequential shift register whose input bit is a linear function of its previous state, i.e. with combinational logic that causes it to pseudo-randomly cycle through a sequence of binary values [4, 5]. A binary feedback register is thus a mapping $\mathfrak{F} : \mathbb{F}_2^n \to \mathbb{F}_2^n$, with $\mathbb{F}_2^n$ the cector space of all binary n-tuples, and $\mathfrak{F}$ of the form:

$$\mathfrak{F} : (x_1, x_2, ..., x_n) \mapsto (x_2, x_3, ..., x_n, f((x_1, x_2, ..., x_n))$$

With f the feedback function, a Boolean operation of n variables, linear in the case of LFSRs. Design-speaking, this function defines the *taps*, i.e. the bits of the register used in the linear operation (generally *XOR*) generating the new input bit [6, 5].

The feedback function can also be be expressed in finite field arithmetic as a polynomial mod 2, the feedback polynomial or reciprocal characteristic polynomial. For instance, a LFSR with taps on the 16th, 14th, 13th and 11th bits has for feedback polynomial $x_{16} + x_{14} + x_{13} + x_{11} + 1$ [7][5].

To obtain a maximal-length LFSR, i.e. a LFSR generating a bit sequence of period $2^n - 1$, several conditions must be met by the feedback function, such as having an even number of taps or using a *relatively-prime* set of taps [7][5].

Algorithm 3.1 details a solution for the generation of linear feedback shift registers, given a set of taps and an initial value.

Algorithm 3.1 Implementation of a generic LFSR

Require:

taps: Sequence of taps defining the register (ex: $(1, 0, 0, 1)$ represents the register of feedback polynomial $x^4 + x^3 + 1$).
seed: Initial value given to the register.

Ensure:

Sequence of values generated by the LFSR defined by the given sequence of taps and initial value.

```
degree ← |taps|                              ▷ Degree of the feedback polynomial
period ← 2^degree − 1                        ▷ Max-period of the LFSR
value ← seed                                 ▷ Returned value
it ← 0
while it < period do
        ▷ Computing first the new value of the most-significant bit:
    bit ← 0
    j ← 0
    for j < degree do
        if taps[j] ≠ 0 then
            bit ← bit ⊕ (value >> j)     ▷ ⊕ representing the binary XOR-operation
        end if
        j ← j + 1
    end for
        ▷ Getting the final value in the register by popping the less-significant bit
        and appending the new most-significant one:
    value ← (value >> 1)|(bit << (degree − 1))
    return value                             ▷ Yield the pseudo-random value
end while
```

NLFSR: The theory of *Non-Linear Feedback Shift Registers* (NLFSR) are the same as for LFSRs, with the only difference than the feedback function f is defined as non-linear[5]. This difference makes NLFSRs harder to predict than LFSRs, but also imposes extensive carefulness in the selection of the feedback function, in order to ensure a maximal period of $2^n - 1$ bits. Conditions and lists of valid configurations can be found in [8].

Algorithm 3.2 shows how to generate non-linear feedback shift registers, given an initial configuration.

3.3 Applications and Uses

Random number generators have applications in every area where unpredictable behavior is desirable or required, from cryptographic systems to gambling applications, statistical sampling, simulation or tests suites, etc. [2, 4].

Depending on the applications, various properties can be required from the generators. For instance, a security application will need a *cryptographically-secure* generator; while a shuffling method will require a generator ensuring the uniqueness of the returned values [2]. While cryptography and some numerical algorithms demand for *qualities* mostly found in physical RNGs, there is a vast variety of computer applications which are satisfied with *weaker* forms of randomness; for example to simulate random behaviors in games or to get an input for a data-integrity checksum. In those cases, PNRGs are often promoted, since they are generally faster and lighter (simple boolean logic for LFSRs and NLFSRs for instance) [2, 4].

4 Testing Randomness

As the definition of *randomness* is complex and field-dependent, so are the tests designed to evaluate the quality of RNGs.

4.1 About the Difficulty to Test Randomness

As exposed in the definition of *Randomness* in Section 2, The term *random sequence* can have various meanings depending on the field of study, making this property difficult to test. Moreover, as for cryptographic results, the large number of possibilities is generally impossible to be fully covered. For instance, testing if a random sequence has indeed *no shorter construction* is impossible without checking every construction [9]. This is why the randomness of a sequence is commonly analyzed through statistical tests or complexity evaluations.

Once a RNG has been developed, a battery of empirical statistical tests is run against it, in an attempt to identify statistical bias. They try to evaluate if the generated sequences follow the hypothesis of *perfect behavior* (named $\mathcal{H}_0$ in [10]), the hypothesis that the values of the sequences " *imitate independent random variables from the uniform distribution*" [10]. Different tests will thus detect different problems by using checking various statistical behaviors. Since a full coverage is impossible, there is no universal battery of tests. It is commonly acknowledged that *good* RNGs are then those which pass *complicated* or *numerous* tests [9, 10].

4.2 Notorious Tests

DIEHARD Tests: This battery of statistical tests has been developed by George Marsaglia and first published in 1995 on a downloadable CD-ROM of random numbers [11]. It consists of fifteen tests, which are run over a large file containing the sequence, provided by the user. Those tests are namely the: birthday spacings, overlapping permutations, ranks of 31x31 and 32x32 matrices, ranks of 6x8 matrices, monkey tests on 20-bit Words, monkey tests OPSO, OQSO, DNA, 1's count in a stream of bytes, 1's count in specific bytes, parking lot, minimum distance, random spheres, squeeze, overlapping sums, runs, and craps [12].

TestU01 Suite: This software library, initiated in 1985 and implemented in the ANSI C language offers a collection of utilities for the empirical statistical testing of RNGs. Among the various provided tools, it offer general implementations of the classical statistical tests for random number generators, several others proposed in the literature, and some original ones; but it also offers tools to implement specific statistical tests. [10].

Berlekamp-Massey Algorithm: This algorithm can't really be considered as a test. It is is used for finding in a arbitrary field $\mathbb{F}_n$ the minimal polynomial of linearly recurrent sequences, such as thoses generated by LFSRs. *Elwyn Berlekamp* invented an algorithm in 1967 for decoding BCH codes [13], then soon after *James Massey* simplified and adapted it to LFSRs [14].

The goal of the algorithm is to determine the minimal degree L and the *annihilator* (or *inverse feedback*) polynomial $F(x)$ of the given sequence S, such as for all syndromes $n = L$ to $(|S| - 1)$:

$$S_n + F_1.S_{n-1} + ... + F_L.S_{n-L} = 0$$

Finding this minimal polynomial requires to solve a set of L linear equations, $F(x)$ being thus uniquely determined by the first $2L$ elements of S. At each iteration l, the algorithm then evaluates the *discrepency* β [5, 15].

$$\beta = S_l + F_1.S_{l-1} + ... + F_L.S_{l-L}$$

If $\beta = 0$, $F(x)$ and L are still currently correct, and the algorithm can proceed to the next iteration. If $\beta \geq 0$, $F(x)$ should be concordantly adjusted, by shifting and scaling the syndromes added since the last update of L [5].

$$F(x) \leftarrow F(x) - (\beta/\alpha).x^{\delta}.f(x)$$

with δ the number of iterations since the last update of L, and α, resp. $f(x)$, the value of β, resp. $F(x)$, before this last update.

In order to keep track of the number of errors and current degree of the polynomial, L should be updated if the number l of iterations is getting bigger then $2L$ (cf. above). In this case, L is adjusted as follow:

$$L \leftarrow l + 1 - L$$

A more complete definition of the process is given in Algorithm 4.1, based on the works in [5, 16]. A Python implementation of this algorithm has been run against the sequences generated by the previously-presented LFSR and NLFSR implementations (Algorithms 3.1 and 3.2). As expected, it was able to successfully and efficiently evaluate the annihilator polynomial of complex LFSRs (and thus their original feedback polynomial, by taking the inverse of the annihilator); and as expected it failed against the sequences generated by the non-linear registers.

5 Conclusion

This paper presented a short overview of the large topic which is the generation of random numbers. For more detailed explanations, please refer to the list of documents below.

References

1. Downey, R.: Some recent progress in algorithmic randomness. In Mathematical Foundations of Computer Science 2004. Springer Berlin Heidelberg (2004)
2. Wikipedia: Random Number Generation (2014)
3. Aumasson, J.P.: The Search and Construction of Nonlinear Feedback Shift Registers. AppSec Forum Switzerland 2013 (2013)
4. Raymond, S., Andrew, S., Patrick, C., Jason, M.: Linear Feedback Shift Register (2001)
5. Joux, A.: Algorithmic cryptanalysis. CRC Press (2009)
6. Szmidt, J.: The Search and Construction of Nonlinear Feedback Shift Registers. Military Communication Institute, Zegrze, Poland (2013)
7. Wikipedia: Linear Feedback Shift Register (2014)
8. Dubrova, E.: A List of Maximum Period NLFSRs. IACR Cryptology ePrint Archive **2012** (2012) 166
9. Ritter, T.: Randomness Tests: A Literature Survey (2007)
10. L'Ecuyer, P, S.R.: TestU01 - A Software Library in ANSI C for Empirical Testing of Random Number Generators (2002)
11. Marsaglia, G.: The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness (2005)
12. Soto, J.: Statistical Testing of Random Number Generators. Proceedings of the 22nd National Information Systems Security Conference **NIST, 1999** (1999) 12
13. Berlekamp, E.R.: Nonbinary BCH decoding. University of North Carolina. Department of Statistics (1967)
14. Massey, J.L.: Shift-register synthesis and BCH decoding. Information Theory. IEEE Transactions **15**(1) (1969) 122–127
15. Feng, G.-L., T.K.: A generalization of the Berlekamp-Massey algorithm for multisequence shift-register. Information Theory, IEEE Transactions **37**(5) (2012) 1274–1287
16. Rodrigez, S.: Implementation of a decoding algorithm for codes from algebraic curves in the programming language Sage. diploma thesis, Faculty of San Diego State University (2013)

Algorithm 3.2 Implementation of a generic NLFSR

Require:

taps: Sequence of combinations of taps defining the non-linear register. (ex: $([0], [1], [2], [1, 2])$ represents the register of feedback polynomial $x^4 + x^3 + x^2 + x^1 * x^2 + 1$).

seed: Initial value given to the register.

Ensure:

Sequence of values generated by the NLFSR defined by the given sequence of taps and initial value.

$degree \leftarrow |taps|$ ▷ Degree of the feedback polynomial
$period \leftarrow 2^d egree - 1$ ▷ Max-period of the NLFSR, cf [8]
$value \leftarrow seed$ ▷ Returned value
$it \leftarrow 0$
while $it < period$ **do**
▷ Computing first the new value of the most-significant bit:
  $bit \leftarrow 0$
  $j \leftarrow 0$
  for $j < degree$ **do**
    if $taps[j] \neq 0$ **then**
▷ Computing the binary multiplication $x_{k_0} \otimes x_{k_1} \otimes ... \otimes x_{k_n}$ with $[k_0, k_1, ..., k_n]$ the j-th taps array
      $multResult \leftarrow 1$
      for all $k \in taps[j]$ **do**
▷ Binary multiplication of terms returns 1 iif none of the terms is null. So if we encounter a null bit, we simply return 0, else 1.
        if $(value >> k) = 0$ **then**
          $multResult \leftarrow 0$
          break
        end if
      end for
    else
      $multResult \leftarrow 0$
    end if
    $bit \leftarrow bit \oplus multResult$ ▷ Binary addition of the multiplication results
    $j \leftarrow j + 1$
  end for
▷ Getting the final value in the register by popping the less-significant bit and appending the new most-significant one:
  $value \leftarrow (value >> 1)|(bit << (degree - 1))$
  return $value$ ▷ Yield the pseudo-random value
end while

Algorithm 4.1 Implementation of the Berlekamp-Massey Algorithm over $\mathbb{F}_2$

Require:

sequence: Sequence of $\mathbb{F}_2$ elements to analyze.

Ensure:

Assumed inverse feedback polynomial.

▷ Note: The variables names are based on those in the pseudo-code found in [5]

$N \leftarrow |sequence|$ ▷ Length of the sequence

$F(x) \leftarrow 1, f(x) \leftarrow 1$ ▷ Polynomials, with F being the supposed inverse feedback polynomial

$L \leftarrow 0$ ▷ Current number of assumed errors

$\delta \leftarrow 1$ ▷ Number of iterations since last update of L

$l \leftarrow 0$ ▷ Computing $F(x)$ and L:

for $l < N$ **do**

  $\beta \leftarrow \sum_{i=0}^{L} sequence[l-i] \otimes |x^i|F$ ▷ Evaluating the current discrepency

  ▷ If the discrepancy is null, we can assume that F and L are currently correct and can continue with the next term. Else, we must adjust F:

  if $\beta \neq 0$ **then** ▷ Adjusting F for this term:

    $g(x) \leftarrow F(x)$

    $F(x) \leftarrow F(x) - x^{\delta} f(x)$

    ▷ The generic algorithm in $\mathbb{F}_q$ requires to multiply the subtractor term by (β/α) with α the previous non-null value of β updated at the same time as L and $f(x)$. In $\mathbb{F}_2$, this product would always be equal to 1.

    if $2 * L \leq l$ **then**

      ▷ L represents the number of error, so the discrepancies will reach 0 before l grows bigger than $2 * L$. If it is not the case, we must update L (and thus re-initalize δ), and also f and α:

      $L \leftarrow l + 1 - L$ ▷ Number of available syndromes used to calculate discrepancies

      $\delta \leftarrow 1$

      $f(x) \leftarrow g(x)$ ▷ $f(x)$ get the previous value of $F(x)$

    else

      $\delta \leftarrow \delta + 1$

    end if

  else

    $\delta \leftarrow \delta + 1$

  end if

  $l \leftarrow l + 1$

end for

return F(x), L