A brief overview of pseudo-random number generators and testing of our own simple generator

Jakub Łukasiewicz

https://orcid.org/0000-0002-4938-504X

May 3, 2022

Abstract

Most random numbers used in computer programs are pseudorandom, which means they are generated in a predictable fashion using a mathematical formula. This is acceptable for many purposes, sometimes even desirable.

In this paper we will take a look at few popular generators producing pseudorandom integers from continuous uniform distribution. Then we will use such generator to try to implement a generator producing numbers from interval $]0, 1[$. And then, on its basis, generators of numbers from Bernoulli, binomial, Poisson, exponential and normal distributions.

Keywords: overview, pseudo, random, number, generator, testing

Contents

1 Introduction

Random numbers generators (RNGs) are needed for practically all kinds of computer applications, such as simulation of stochastic systems, numerical analysis, probabilistic algorithms, secure communications, computer games, and gambling machines, to name a few. [12]

One way of achieving randomness is by using entropy of the "outside" world. For example, in the 1940's one could get a large deck of punched cards filled with random sampling digits. Those cards could be placed in the data section of a program. [10]
However for many use cases reading random numbers from other external storage devices was too slow and the size of main memory was much too limited to store large tables of random digits. Thus two types of solutions emerged to produce random numbers on the fly, in real time:

- using a fast physical device that produces/collects random noise
- a purely deterministic algorithm producing a sequence *imitating* randomness.

2 Definition

Pseudorandom number generator [1] (PRNG) – a deterministic algorithm that has one or more inputs called "seeds", and it outputs a sequence of values that appears to be random. [8]

2.1 Mathematical definition (L'Ecuyer)

A *generator* is a structure $\mathcal{G} = (S, s_0, T, U, G)$, where S is a finite set of *states*, $s_0 \in S$ is the *initial state* (*seed*), $T : S \to S$ is the *transition function*, U is a finite set of *output* symbols and $G : S \to U$ is the *output function*. A generator operates as follows:

Start from the seed s_0 and let $u_0 := G(s_0)$. Then, for $i := 1, 2, \ldots$ let $s_i = T(s_{i-1})$ and $u_i = G(s_i) \in U$. We assume that efficient procedures are available to compute T and G. The sequence $\{u_i\}$ is the output of the generator and its elements are called the *observations*. For pseudorandom number generators, one would expect the observations to behave from the outside as if they were the values of independent and identically random variables, uniformly distributed over U. The set U is often a set of integers of the form $\{0, \ldots, m-1\}$ or a finite set of values between 0 and 1 to approximate the $U(0, 1)$ distribution. [11]

Period and transient

Since S is finite, the sequence of states is ultimately periodic. The *period* is the smallest positive integer ρ such that $s_{\rho+n} = s_n$ for some integer $\tau \geqslant 0$ and for all $n \geqslant \tau$. The smallest τ with this property is called *transient*. When $\tau = 0$, the sequence is said to be *purely periodic*. [11]

3 Examples of existing PRNG

3.1 *Middle-square method*

The method was invented by John von Neumann, and was described at a conference in 1949. [21]

To generate a sequence of n-digit pseudorandom numbers, an n-digit seed is created and squared, producing a $2n$-digit number. If the result has fewer than $2n$ digits, leading zeroes are added to compensate. The middle n digits of the result would be the next number in the sequence, and returned as the result. This process is then repeated to generate more numbers. [21, 27]

[1] also referred to as: *deterministic random bit generator* (DRBG) [8]

3.2 *Linear congruential generator* (LCG)

By far one of the most popular random number generators in use today are special cases of the following scheme, introduced by D. H. Lehmer in 1949. [14]
As we read in [9], to create LCG we need four integers:

- the modulus m $(0 < m)$
- the multiplier a $(0 \leqslant a < m)$
- the increment c $(0 \leqslant c < m)$
- the seed X_0 $(0 \leqslant X_0 < m)$

The desired sequence of random numbers is then obtained by setting:

$$X_{n+1} = (a \cdot X_n + c) \mod m \tag{1}$$

3.2.1 *Lehmer random number generator*

The special case of (1) with $c = 0$ deserves explicit mention, since it's Lehmer's original method (and the number generation process is a little faster [9]).

$$X_{k+1} = a \cdot X_k \mod m \tag{2}$$

The terms *multiplicative congruential method* and *mixed congruential method* are used by many authors to denote linear congruential sequences respectively with $c = 0$ and $c \neq 0$.

3.3 *Lagged Fibonacci generator* (LFG)

Fibonacci Generators is a class of random number generator aimed at being an improvement on the "standard" linear congruential generator. These are based on a generalisation of the Fibonacci sequence, hence the formula:

$$X_n = (X_{n-1} + X_{n-2}) \mod m \quad \leftrightarrow \; n \geqslant 2$$

Fibonacci Generator has good quality compared to other linear generators, but requires much more computations. The disadvantage of this generator are high correlations between the elements of the sequence. The sequences satisfy the decomposition condition but do not satisfy the independence condition. This disadvantage can be eliminated by generalizing the formula to a form called *lagged Fibonacci generator* (LFG):

$$X_n = (X_{n-p} \diamond X_{n-q}) \mod m \quad \leftrightarrow \; n \geqslant p > q \geqslant 1 \tag{3}$$

where $\diamond$ is some mathematical operator (e.g. addition, subtraction, XOR). [6]

3.4 *Linear-feedback shift register* (LFSR)

LFSR is a shift register whose input bit is a linear function of its previous state.
Robert C. Tausworthe in 1965 defined [10, 31] LFSR generator via

$$\begin{aligned} X_n &= (a_1 X_{n-1} + \cdots + a_k X_{n-k}) \mod 2 \\ u_i &= \sum_{l=1}^{w} \frac{X_{is+l-1}}{2^l} \end{aligned} \tag{4}$$

where $a_1, \ldots, a_k \in \mathbb{F}_2, a_k = 1$ ($\mathbb{F}_2$ is *Galois field*) and w, s are positive integers. It takes a block of w successive bits every s steps of the linear recurrence and constructs the output u_i from that.

3.4.1 *Generalized feedback shift register* (GFSR)

GFSR generator [15] is a widely used pseudorandom number generator based on the linear recurring equation:

$$X_{l+n} = X_{l+m} \oplus X_l \qquad \leftrightarrow \; l \geqslant 0 \tag{5}$$

where each X_l is a word with components 0 or 1 of size w, and $\oplus$ denotes bitwise exclusive-or operation. [18]

3.5 ACORN

***A**dditive **Co**ngruential **R**andom **N**umber* (ACORN) generator, introduced by R.S. Wikramaratna [35], was originally designed for use in geostatistical and geophysical Monte Carlo simulations, and later extended for use on parallel computers. [36]

We define [36] the kth order ACORN generator X_j^k recursively from a seed X_0^0 (where $0 < X_0^0 < M$ and $M = 1, 2, \dots$) and a set of k initial values X_0^m (where $m = 1, \dots, k$ and $0 \leqslant X_0^m < M$) by:

$$\begin{aligned} X_n^0 &= X_{n-1}^0 && \leftrightarrow \; n \geqslant 1 \\ X_n^m &= (X_n^{m-1} + X_{n-1}^m) \bmod M && \leftrightarrow \; n \geqslant 1, m = 1, \dots, k \end{aligned} \tag{6}$$

> The interested are encouraged to visit the official website [36].

3.6 *Mersenne Twister* (MT)

It is by far the most widely used general-purpose PRNG. Its name derives from the fact that its period length is chosen to be a Mersenne prime.
MT was developed in 1997 by Makoto Matsumoto and Takuji Nishimura as a new variant of the *twisted GFSR* (TGFSR) [19].
The most commonly used version of the Mersenne Twister algorithm is based on the Mersenne prime $2^{19937} - 1$. The standard implementation of that, MT19937, uses a 32-bit word length.

> Due to the size, the mathematical definition will be omitted. The interested are encouraged to read the original paper [20] and to visit the official website [17].

3.7 *Xorshift*

Xorshift RNGs are a class of PRNGs discovered by George Marsaglia. [16]. They are a subset of LFSRs which allow a particularly efficient implementation in software without using excessively sparse polynomials. They generate the next number in their sequence by repeatedly taking the exclusive or of a number with a bit-shifted version of itself.

> The original paper [16] does not contain a straightforward mathematical definition. The interested are encouraged to read also [22] and [2]. In place of mathematical definition an example based on the implementation provided by the original paper [16] will be presented.

```
uint32_t xorshift32() {
    static uint32_t x = 2463534242;
    x ^= (x << a); x  = (x >> b);
    return (x ^= (x << c));
}
```

```
uint32_t xorshift32() {
    static uint32_t x = 2463534242;
    x ^= (x << 13); x  = (x >> 17);
    return (x ^= (x << 5));
} // a,b,c = 13,17,5
```

4 Own generator

Without putting much thought into it let us make our generator to be a combination of LCG, LFG and Xorshift.
First let us combine (1) and (3) into:

$$X_n = \big(a(X_{n-p} \diamond X_{n-q}) + c\big) \mod m \quad \leftrightarrow \; n \geqslant p > q \geqslant 1 \tag{7}$$

Now we need to handle the $\diamond$. Let it be $+$ for even X_{n-q} and $\oplus$ for X_{n-q} odd. Thus (7) transforms into:

$$X_n = \begin{cases} \big(a(X_{n-p} + X_{n-q}) + c\big) \mod m & \leftrightarrow \; 2|X_{n-q} \\ \big(a(X_{n-p} \oplus X_{n-q}) + c\big) \mod m & \leftrightarrow \; 2 \nmid X_{n-q} \end{cases} \leftrightarrow \; n \geqslant p > q \geqslant 1 \tag{8}$$

Let us use the Marsaglia's favourite values of Xorshift in (8) too (with $m = 2^b$), thus:

$$X_n = \begin{cases} \big(13(X_{n-p} + X_{n-q}) + 5\big) \mod 2^{17} & \leftrightarrow \; 2|X_{n-q} \\ \big(13(X_{n-p} \oplus X_{n-q}) + 5\big) \mod 2^{17} & \leftrightarrow \; 2 \nmid X_{n-q} \end{cases} \leftrightarrow \; n \geqslant p > q \geqslant 1 \tag{9}$$

The remaining variables are p and q. Let $p = 7$ and $q = 3$, thus pre-final formula is:

$$X_n = \begin{cases} \big(13(X_{n-7} + X_{n-3}) + 5\big) \mod 2^{17} & \leftrightarrow \; 2|X_{n-q} \\ \big(13(X_{n-7} \oplus X_{n-3}) + 5\big) \mod 2^{17} & \leftrightarrow \; 2 \nmid X_{n-q} \end{cases} \leftrightarrow \; n \geqslant 7 \tag{10}$$

Initial values will generate Xorshift with a "twist" that seed $s \leftarrow s + (s \mod 1000) \cdot b$

In the end the generator looks like:

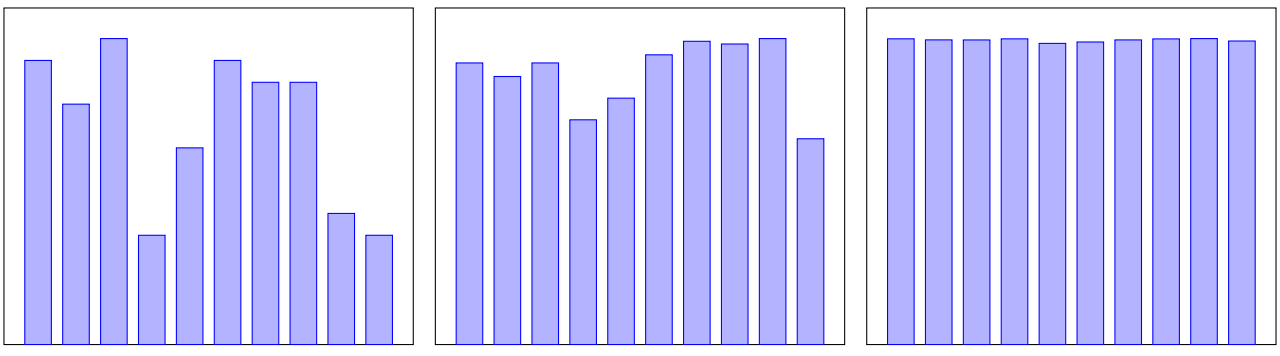
$$X_n = \begin{cases} \text{Xorshift}(s + (s \mod 1000) \cdot b) & \leftrightarrow \; n < 7 \\ \left.\begin{array}{ll} \big(13(X_{n-7} + X_{n-3}) + 5\big) \mod 2^{17} & \leftrightarrow \; 2|X_{n-q} \\ \big(13(X_{n-7} \oplus X_{n-3}) + 5\big) \mod 2^{17} & \leftrightarrow \; 2 \nmid X_{n-q} \end{array}\right\} & \leftrightarrow \; n \geqslant 7 \end{cases} \tag{11}$$

5 Distributions

5.1 Uniform

As operation $\mod m$ is being used in our generator, thus none of random numbers X will be greater than m. Thus to obtain an uniform distribution on interval $]0, 1[$ we need to just divide the result of generator (11) by value m. [9]

$$U_n = \frac{X_n}{m} \tag{12}$$

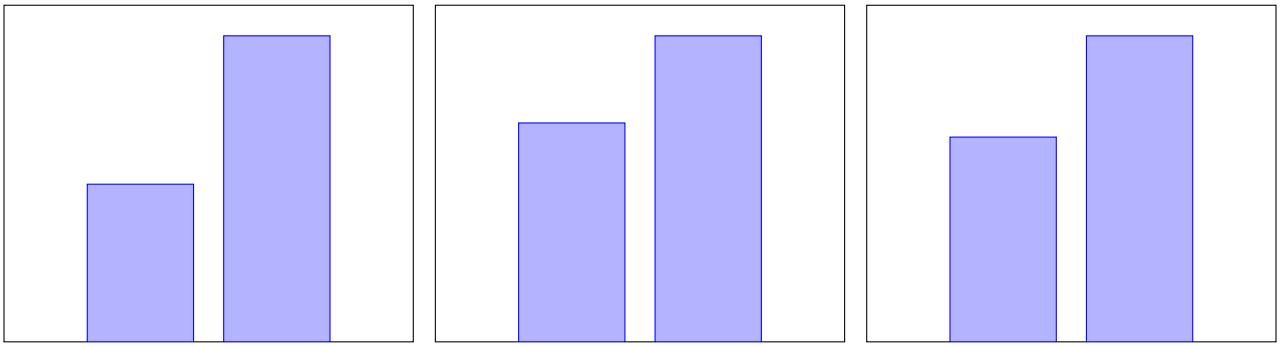


$N = 100$ $N = 1000$ $N = 100000$

5.2 Bernoulli

Bernoulli distribution is a discrete probability distribution of a random variable which takes the value 1 with probability p and the value 0 with probability $1-p$. We will use for this the previously defined uniform distribution (12).

$$B_n(p) = \begin{cases} 0 & \leftrightarrow \ U_n > p \\ 1 & \leftrightarrow \ U_n \leqslant p \end{cases} \tag{13}$$



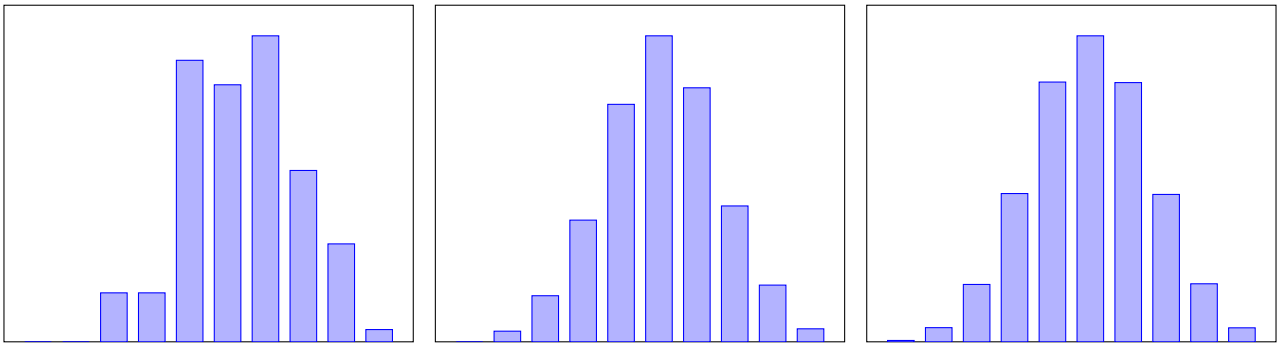
$N = 100$ $N = 1000$ $N = 100000$

$p = 0.6$

5.3 Binomial

Binomial distribution with parameters p, K is the discrete probability distribution of the number of successes in a sequence of K Bernoulli trials (13). [26]

$$B'_n(p,k) = \sum_{i=1}^{K} B_i(p) \tag{14}$$



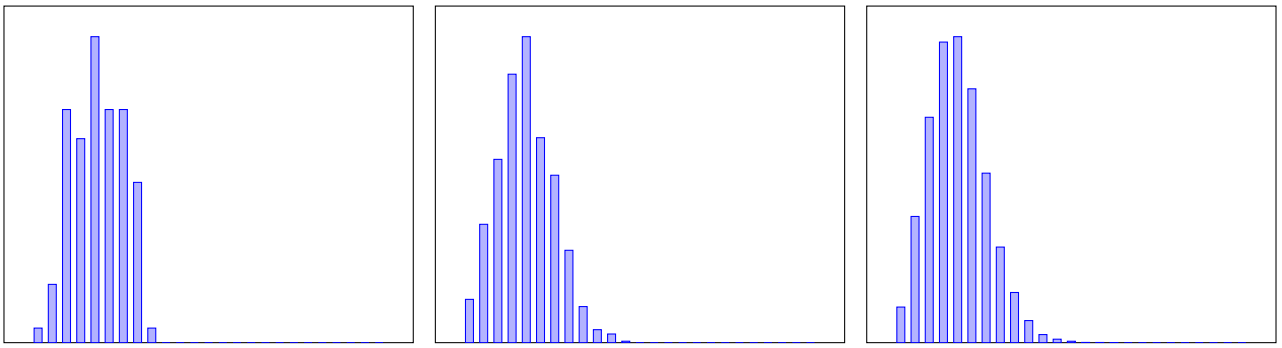
$N = 100$ $N = 1000$ $N = 100000$

$p = 0.5, \ K = 10$

5.4 Poisson

Knuth's algorithm [5, 9]

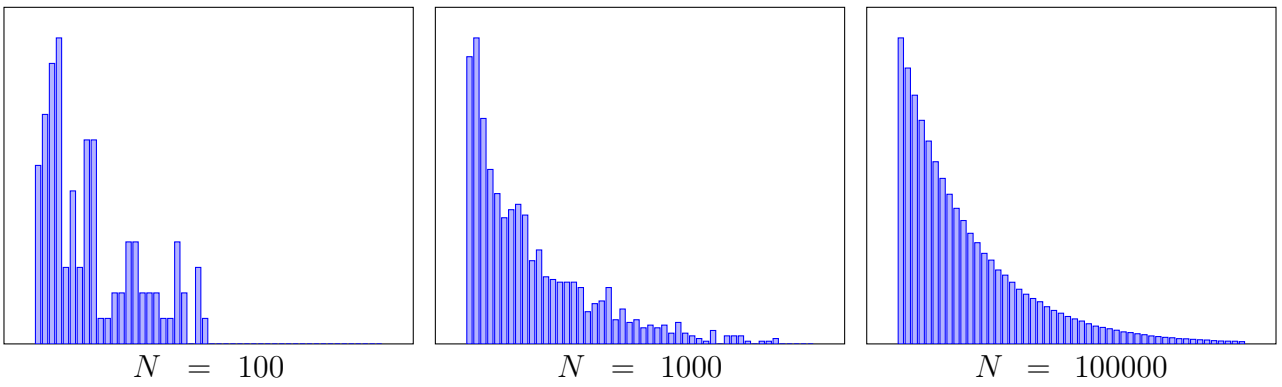
$L \leftarrow \mathrm{e}^{-\lambda}, \quad k \leftarrow 0, \quad p \leftarrow 1$
repeat
  $k \leftarrow k + 1$
  $p \leftarrow p \cdot u()$ { where $u()$ returns uniform random number in $]0, 1[$ }
until $p \leqslant L$
return $k - 1$



5.5 Exponential

To generate exponentially distributed number we will use yet again uniform one [28, 4]

$$E_n(\lambda) = \frac{\ln(1 - U_n)}{-\lambda} \tag{15}$$

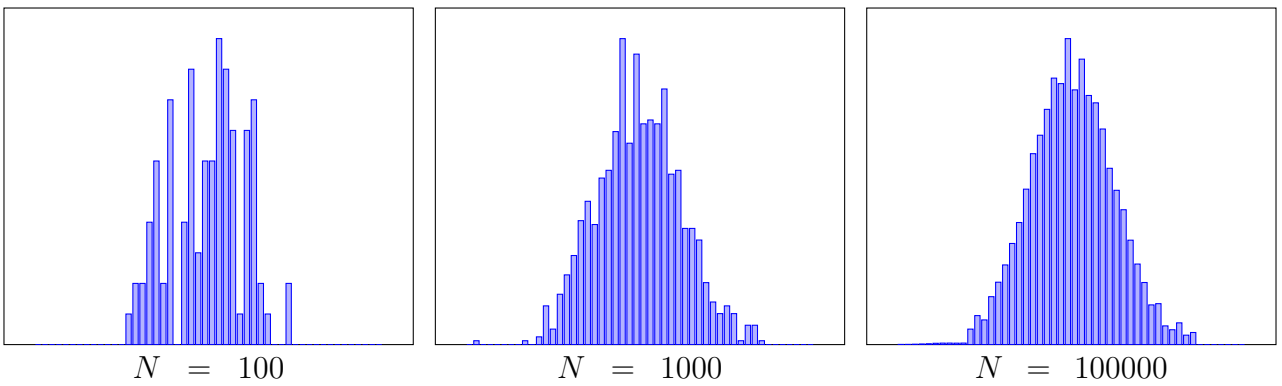


5.6 Normal

One of the commonly used methods to generate Gaussian-distributed numbers from a regular random generator is Box-Muller transform. The method uses two independent random numbers U and V distributed uniformly on $]0, 1[$. Then the two random variables Y_1 and Y_2

$$Y_1 = \sqrt{-2 \ln U} \cos 2\pi V \qquad Y_2 = \sqrt{-2 \ln U} \sin 2\pi V \tag{16}$$

One is returned and the other saved for the next request for a random number. [30]



6 Implementation in C++11

Combining informations from sections 4 and 5 to write an implementation of our PRNG in C++11:

```
class PRNG {

    using base_type = uint32_t;
    base_type X[7]; // for convenience, 'p' is defined here as array size!
    std::size_t N;
    base_type a, b, c, m, p, q;
    double nextNormal;

public:

    PRNG(base_type seed = 2463534242) :
        N(sizeof(X)/sizeof(X[0])),
        a(13), b(20), c(5), m(1 << b), p(N), q(3),
        nextNormal(0)
    {
        seed += (seed % 1000) * b; // "twist"
        for (auto& x: X) { // Xorshift
            seed ^= (seed << a);
            seed  = (seed >> b);
            x = (seed ^= (seed << c));
        }
    }

    base_type operator ()()
    {
        auto P = (N-p) % p;
        auto Q = (N-q) % p;
        if (X[Q] % 2 == 0) {
            X[P] = (a*(X[P] + X[Q]) + c) % m;
        }
        else {
            X[P] = (a*(X[P] & X[Q]) + c) % m;
        }
        return X[P];
    }

    double uniform() { return static_cast<double>( (*this)() ) / m; }

    bool bernoulli(double P) { return uniform() <= P ? 1 : 0; }

    double exponential(double l) { return std::log(1 - uniform()) / -l; }

    std::size_t binomial(double P, std::size_t n)
    {
        auto val = 0;
        while (n--) {
            val += bernoulli(P);
        }
        return val;
    }
```

```
    base_type poisson(double l)
    {
        double L = std::exp(-l);
        base_type k = 0;
        double p = 1;
        do {
            ++k;
            p *= uniform();
        } while (p > L);
        return k-1;
    }

    double normal()
    {
        if (nextNormal != 0) {
            auto temp = nextNormal;
            nextNormal = 0;
            return temp;
        }
        auto r = std::sqrt(-2 * std::log(uniform()));
        auto s = 2 * M_PI * uniform();
        nextNormal = r * std::sin(s) + DBL_MIN;
        return r * std::cos(s);
    }

};
```

7 Tests

7.1 Diehard

10^8 samples (∼6.6 GiB) in formatted [32] data file.
Command: `dieharder -a -g 202 -f out/data/testing.dat`
Unfortunately, the generator seems to fail Dieharder tests.
Process has been manually terminated after 58th FAIL.
Generated output:

```
#==================================================================#
#            dieharder version 3.31.1 Copyright 2003 Robert G. Brown          #
#==================================================================#
   rng_name    |           filename             |rands/second|
     file_input|            out/data/testing.dat|  1.76e+06  |
#==================================================================#
        test_name   |ntup| tsamples |psamples|  p-value |Assessment
#==================================================================#
   diehard_birthdays|   0|       100|     100|0.00000000|  FAILED
      diehard_operm5|   0|   1000000|     100|0.00000000|  FAILED
  diehard_rank_32x32|   0|     40000|     100|0.00000000|  FAILED
    diehard_rank_6x8|   0|    100000|     100|0.00000000|  FAILED
   diehard_bitstream|   0|   2097152|     100|0.00000000|  FAILED
        diehard_opso|   0|   2097152|     100|0.00000000|  FAILED
        diehard_oqso|   0|   2097152|     100|0.00000000|  FAILED
```

```
         diehard_dna|   0|   2097152|     100|0.00000000|  FAILED
diehard_count_1s_str|   0|    256000|     100|0.00000000|  FAILED
diehard_count_1s_byt|   0|    256000|     100|0.00000000|  FAILED
 diehard_parking_lot|   0|     12000|     100|0.00000000|  FAILED
    diehard_2dsphere|   2|      8000|     100|0.00000000|  FAILED
    diehard_3dsphere|   3|      4000|     100|0.00000000|  FAILED
     diehard_squeeze|   0|    100000|     100|0.00000000|  FAILED
        diehard_sums|   0|       100|     100|0.00000000|  FAILED
        diehard_runs|   0|    100000|     100|0.00000000|  FAILED
        diehard_runs|   0|    100000|     100|0.00000000|  FAILED
       diehard_craps|   0|    200000|     100|0.00000000|  FAILED
       diehard_craps|   0|    200000|     100|0.00000000|  FAILED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED
 marsaglia_tsang_gcd|   0|  10000000|     100|0.00000000|  FAILED
         sts_monobit|   1|    100000|     100|0.00000000|  FAILED
            sts_runs|   2|    100000|     100|0.00000000|  FAILED
          sts_serial|   1|    100000|     100|0.00000000|  FAILED
          sts_serial|   2|    100000|     100|0.00000000|  FAILED
          sts_serial|   3|    100000|     100|0.00000000|  FAILED
          sts_serial|   3|    100000|     100|0.00000000|  FAILED
          sts_serial|   4|    100000|     100|0.00000000|  FAILED
          sts_serial|   4|    100000|     100|0.00000000|  FAILED
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   5|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   6|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   7|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   8|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|   9|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.00000000|  FAILED
          sts_serial|  10|    100000|     100|0.00000000|  FAILED
          sts_serial|  11|    100000|     100|0.00000000|  FAILED
          sts_serial|  11|    100000|     100|0.00000000|  FAILED
          sts_serial|  12|    100000|     100|0.00000000|  FAILED
          sts_serial|  12|    100000|     100|0.00000000|  FAILED
          sts_serial|  13|    100000|     100|0.00000000|  FAILED
          sts_serial|  13|    100000|     100|0.00000000|  FAILED
          sts_serial|  14|    100000|     100|0.00000000|  FAILED
          sts_serial|  14|    100000|     100|0.00000000|  FAILED
          sts_serial|  15|    100000|     100|0.00000000|  FAILED
          sts_serial|  15|    100000|     100|0.00000000|  FAILED
          sts_serial|  16|    100000|     100|0.00000000|  FAILED
          sts_serial|  16|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   1|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   2|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   3|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   4|    100000|     100|0.00000000|  FAILED
         rgb_bitdist|   5|    100000|     100|0.00000000|  FAILED
```

7.2 ENT

ENT [34] program, although not ideal for the given input, yelds much more interesting results:

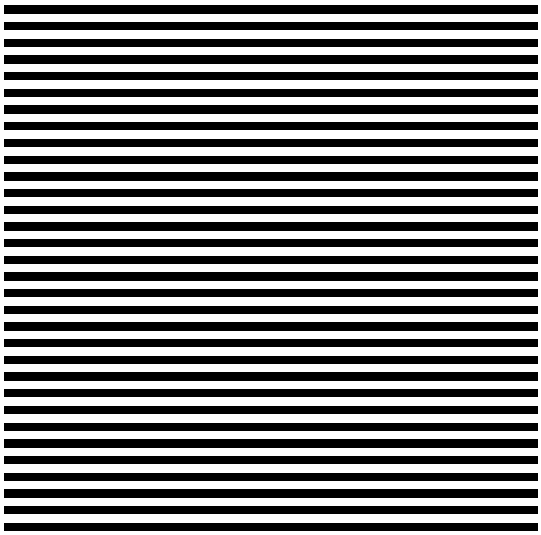
```
Entropy = 3.524750 bits per byte.

Optimum compression would reduce the size
of this 7000000207 byte file by 55 percent.

Chi square distribution for 7000000207 samples is 156174869624.00, and randomly
would exceed this value less than 0.01 percent of the times.

Arithmetic mean value of data bytes is 43.7601 (127.5 = random).
Monte Carlo value for Pi is 4.000000000 (error 27.32 percent).
Serial correlation coefficient is 0.117267 (totally uncorrelated = 0.0).
```

7.3 Visual test



This test brutally shows whats wrong with our generator. It has pattern: numbers alternate between even and odd. The first 10 numbers:

```
18431
82898
3465
19412
95651
38182
77517
64584
27527
7007
```

8 Conclusions

Despite succeeding at creating all desired distributions, we ultimately failed at creating proper pseudorandom number generator. Although on the first glance numbers could appear plausible, it took to our third test – ironically, visual one – to discover its fatal flaw. It just shows that pseudorandomness is no trivial matter and even the simplest algorithms actually have a lot of thought put behind them.

References

[1] adsk. *How to test a random generator*. StackOverflow. URL: https://stackoverflow.com/q/2130621.

[2] Richard P. Brent. "Note on Marsaglia's Xorshift Random Number Generators". In: *Journal of Statistical Software, Articles* 11.5 (2004), pp. 1–5. ISSN: 1548-7660. DOI: 10.18637/jss.v011.i05.

[3] Robert G. Brown. *Dieharder: A Random Number Test Suite*. Duke University Physics Department. URL: https://webhome.phy.duke.edu/~rgb/General/dieharder.php.

[4] Tomasz Chwiej. *Generatory liczb pseudolosowych*. AGH. URL: http://home.agh.edu.pl/~chwiej/mn/generatory_16.pdf (visited on 2021-05-16).

[5] Wikipedia contributors. *Generating Poisson-distributed random variables*. 2020-06-01. URL: https://en.wikipedia.org/wiki/Poisson_distribution#Generating_Poisson-distributed_random_variables (visited on 2021-06-03).

[6] Wikipedia contributors. *Generator Fibonacciego*. 2020-03-28. URL: https://pl.wikipedia.org/wiki/Generator_Fibonacciego (visited on 2021-05-30).

[7] Wikipedia contributors. *Pseudorandom number generator*. 2021. URL: https://en.wikipedia.org/w/index.php?title=Pseudorandom_number_generator (visited on 2021-05-15).

[8] CSRC. *Pseudorandom Number (or Bit) Generator*. URL: https://csrc.nist.gov/glossary/term/Pseudorandom_Number_Generator.

[9] Donald E. Knuth. *The art of computer programming. Seminumerical Algorithms*. 3rd ed. Vol. 2. Addison-Wesley Longman Publishing Co., Inc., 1997. Chap. 3. ISBN: 0201896842. DOI: 10.5555/270146.

[10] Pierre L'Ecuyer. "History of Uniform Random Number Generation". In: *Proceedings of the 2017 Winter Simulation Conference*. IEEE Press, 2017, pp. 202–230. URL: https://www.informs-sim.org/wsc17papers/includes/files/016.pdf.

[11] Pierre L'Ecuyer. "Uniform random number generation". In: *Annals of Operations Research* 53.1 (1994), pp. 77–120. ISSN: 1572-9338. DOI: 10.1007/BF02136827.

[12] Pierre L'Ecuyer and Richard Simard. "TestU01: A C Library for Empirical Testing of Random Number Generators". In: *ACM Trans. Math. Softw.* 33.4 (2007-08). ISSN: 0098-3500. DOI: 10.1145/1268776.1268777.

[13] Lawrence L. Leemis and Stephen K. Park. *Discrete-Event Simulation: A First Course*. 2007, pp. 37–99.

[14] D.H. Lehmer. "Mathematical Methods in Large-scale Computing Units". In: *Proceedings of the Second Symposium on Large Scale Digital Computing Machinery*. 1949, pp. 141–146. URL: https://archive.org/details/proceedings_of_a_second_symposium_on_large-scale_/page/n178.

[15] T. G. Lewis and W. H. Payne. "Generalized Feedback Shift Register Pseudorandom Number Algorithm". In: *J. ACM* 20.3 (1973-06), pp. 456–468. ISSN: 0004-5411. DOI: 10.1145/321765.321777.

[16] George Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software, Articles* 8.14 (2003), pp. 1–6. ISSN: 1548-7660. DOI: 10.18637/jss.v008.i14.

[17] Makoto Matsumoto. *Mersenne Twister Home Page*. URL: http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/emt.html (visited on 2021-05-31).

[18] Makoto Matsumoto and Yoshiharu Kurita. "Twisted GFSR Generators". In: *ACM Trans. Model. Comput. Simul.* 2.3 (1992-07), pp. 179–194. ISSN: 1049-3301. DOI: 10.1145/146382.146383. URL: http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/ARTICLES/tgfsr3.pdf.

[19] Makoto Matsumoto and Yoshiharu Kurita. "Twisted GFSR Generators II". In: *ACM Trans. Model. Comput. Simul.* 4.3 (1994-07), pp. 254–266. ISSN: 1049-3301. DOI: 10.1145/189443.189445. URL: http://www.math.sci.hiroshima-u.ac.jp/m-mat/MT/ARTICLES/ttgfsr7.pdf.

[20] 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. URL: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf.

[21] John von Neumann. "Various Techniques Used in Connection with Random Digits". In: *Monte Carlo Method*. Ed. by A. S. Householder, G. E. Forsythe, and H. H. Germond. Vol. 12. National Bureau of Standards Applied Mathematics Series. Washington, DC: US Government Printing Office, 1951. Chap. 13, pp. 36–38. URL: https://mcnp.lanl.gov/pdf_files/nbs_vonneumann.pdf.

[22] François Panneton and Pierre L'Ecuyer. "On the Xorshift Random Number Generators". In: *ACM Trans. Model. Comput. Simul.* 15.4 (2005-10), pp. 346–361. ISSN: 1049-3301. DOI: 10.1145/1113316.1113319. URL: https://www.iro.umontreal.ca/~lecuyer/myftp/papers/xorshift.pdf.

[23] W. H. Payne, J. R. Rabung, and T. P. Bogyo. "Coding the Lehmer Pseudo-Random Number Generator". In: *Commun. ACM* 12.2 (1969-02), pp. 85–86. ISSN: 0001-0782. DOI: 10.1145/362848.362860.

[24] *Pseudo-random number generation algorithms*. MathOverflow. 2010-06-26. URL: https://mathoverflow.net/q/29494.

[25] *RANDOM.org – Simple Visual Analysis*. URL: https://www.random.org/analysis/#visual.

[26] Edward Ross. "From Bernoulli to Binomial Distributions". In: (). URL: https://skeptric.com/bernoulli-binomial/ (visited on 2021-06-09).

[27] PBS Infinite Series. *How to Generate Pseudorandom Numbers*. 2017. URL: https://youtu.be/C82JyCmtKWg.

[28] Alok Singhal. *Pseudorandom Number Generator - Exponential Distribution*. StackOverflow. URL: https://stackoverflow.com/a/2106564.

[29] Shobhit Sinha, SK Hafizul Islam, and Mohammad S. Obaidat. "A comparative study and analysis of some pseudorandom number generator algorithms". In: *Security and Privacy* 1.6 (2018), 1:e46. DOI: 10.1002/spy2.46.

[30] S.Lott. *Generate random numbers following a normal distribution in C/C++*. StackOverflow. URL: https://stackoverflow.com/a/2325531.

[31] Robert Tausworthe. "Random Numbers Generated by Linear Recurrence Modulo Two". In: *Mathematics of Computation - Math. Comput.* 19 (1965-05), pp. 201–201. DOI: 10.2307/2003345.

[32] Paul Uszak. *How can I make my input file suitable for Dieharder?* Cryptography Stack Exchange. URL: https://crypto.stackexchange.com/a/87122.

[33] John Viega. "Practical Random Number Generation in Software". In: *Proc. 19th Annual Computer Security Applications Conference*. 2003. URL: https://www.acsac.org/2003/papers/79.pdf.

[34] John Walker. *ENT – A Pseudorandom Number Sequence Test Program*. Fourmilab, 2008-01-28. URL: https://www.fourmilab.ch/random.

[35] Roy S. Wikramaratna. "ACORN – A new method for generating sequences of uniformly distributed Pseudo-random Numbers". In: *Journal of Computational Physics* 83.1 (1989), pp. 16–31. ISSN: 0021-9991. DOI: 10.1016/0021-9991(89)90221-0.

[36] Roy S. Wikramaratna. *ACORN random numbers*. 2019-03-31. URL: http://acorn.wikramaratna.org (visited on 2021-05-30).

Source code

All source files (LaTeX, BibTeX, C++, Makefile) are available on GitHub under the URL: https://github.com/Jorengarenar/PRNG-paper