Chapter 1

Random numbers

The generation of random numbers is too important to be left to chance.

ROBERT R. COVEYOU,
OAK RIDGE NATIONAL LABORATORY

Almost all kinds of simulations in physics, chemistry, engineering, economics and social science use random numbers somewhere during the process. Random numbers are also needed in practical applications such as modelling the behavior of stock markets, machine learning and deep learning. Perhaps the most critical one is cryptography: Every time you use a computer or a mobile phone, you are using random numbers. The security of on-line banking, purchases and messaging all depend on random numbers.

A precise mathematical definition of random numbers is difficult to give but let us start with an informal one, which contains the most important properties we demand from random numbers. First, we demand that the random numbers are evenly distributed in interval (0,1); as the preceding indicates, random numbers are assumed to be real. This is also how your computer's intrinsic RNG produces random numbers. The random numbers should be random. How can we define random? Intuitively, an arbitrarily long sequence of numbers should "look" random. That means that no sub-interval within (0,1) should be preferred, and there should be no recurring patterns or sequences of numbers. To be more precise, there should be no correlations between successive random numbers or sequences of of random numbers. We will return to this topic later and define some statistical and physical measures as how to study randomness.

You may ask if it really matters or if it is really so hard to define what is random. As for the first question, it is easy to convince yourself if you think of, for example, a very practical and important problem of data encryption. If your random numbers have predictable correlations or repeating patterns, it may become very easy to break the encryption, which, in practice, could lead, for example, to unsafe transmission of your credit card information with on-line purchases. As for the second question, try to come up with a definition of randomness! Is the sequence $\{1,1,1,1\}$ more random than $\{2190,9,1,14,276\}$? If you picked one over the other, what is your criterion and can it be applied generally?

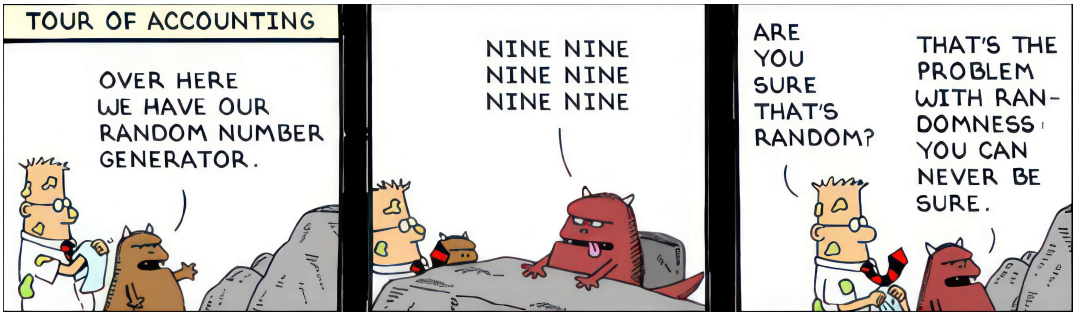

Figure 1.1: As this comic strip from Dilbert points out, randomness is not a trivial matter.

1.1 Random numbers vs Pseudo-random numbers

Let us start from the concept of randomness. *True random numbers* are those that are completely unpredictable and generated in a non-deterministic manner, typically from physical phenomena. Examples include electronic noise, radioactive decay, atmospheric noise, and thermal noise. These phenomena are inherently unpredictable and provide true randomness. random.org [18] utilizes atmospheric noise to generate random numbers. *Pseudorandom numbers* are numbers that appear random but are generated by deterministic computational algorithms. They are not truly random because their generation depends on an initial value, known as a seed. That sounds almost like an oxymoron: how can we produce random numbers using a deterministic algorithm? That is the topic we will focus on.

With the above, we classify random numbers as

1. True random numbers and
2. Pseudorandom numbers.

Those numbers are tabularized and may then be used. The problem is, however, that that method is very impractical. To overcome that difficulty, several algorithms have been developed to produce sequences of pseudo-random numbers which mimic the true random numbers as well as possible. The goodness, i.e., quality, of these pseudo-random numbers can be tested in various ways as will be discussed below. When we discuss random numbers, we actually mean pseudo-random numbers unless otherwise mentioned.

Development of random number generators has been a very active field of research since (and already before) the invent of modern computers and remains to be so even today.

1.1.1 Can humans produce random numbers?

The question of whether humans can produce truly random numbers is a topic that bridges cognitive psychology, neuroscience, and mathematical theory.

Randomness, by definition, implies a lack of pattern, predictability, or discernible order. In mathematical and computational contexts, random sequences are used extensively in areas such as cryptography, statistical sampling, and simulations. These applications often rely on algorithmically generated random numbers, known as pseudorandom numbers, due to their reproducibility and perceived lack of pattern. This computational approach contrasts starkly with the way humans perceive and generate randomness, see Figure 1.1.

Human Cognition and Random Number Generation

Human cognition is inherently pattern-seeking. This tendency is a survival mechanism, aiding in recognizing threats and opportunities in the environment. However, this predisposition is at odds with the concept of generating randomness. Psychological studies have consistently shown that when humans attempt to create random sequences, certain biases and patterns emerge. For instance, individuals might avoid repeating the same number, under the mistaken belief that repetition is less random, or they might over-represent numbers they subconsciously prefer.

One illustrative study by Wagenaar [7] found that when asked to simulate coin tosses, participants produced sequences far from statistically random, avoiding long runs of the same outcome, which are common in truly random sequences. This finding underscores a key aspect of human-generated randomness - it is often influenced by misconceptions about what constitutes randomness.

To objectively assess the randomness of sequences, whether generated by humans or computers, statistical methods like the χ^2-test are employed. These tests often reveal the non-random nature of human-generated sequences. For example, a simple exercise where individuals are asked to create a sequence of random numbers typically results in distributions that significantly deviate from what would be expected in a truly random sequence, showcasing predictable patterns or biases.

In contrast to the limitations of human-generated randomness, algorithmic methods, though not truly random, offer a higher degree of unpredictability and reproducibility, crucial for applications in cryptography and statistical modeling. These methods, based on complex algorithms, can produce sequences that pass rigorous statistical tests for randomness and are free from human cognitive biases.

1.2 Historical tidbits and future directions

As John von Neumann joked, "*Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin.*"

The development of pseudorandom number generators (PRNGs) has been a cornerstone in computational science, influencing fields ranging from cryptography to statistical simulations.

The inception of pseudorandom numbers can be traced back to the early 20th century, driven by the increasing need for 'random-like' sequences in statistics and other scientific computations. The first methods of generating pseudorandom numbers involved mechanical devices like roulette wheels, dice, and shuffled cards. However, these methods were labor-intensive and not suitable for large-scale computations.

One of the earliest algorithmic approaches was John von Neumann's Middle Square Method [2, 8]. It involved squaring a number and taking the middle digits as the next number in the sequence. Despite its simplicity, the method had limitations in terms of cycle length and distribution properties.

The 1950's brought also Linear Congruential Generators, or LCGs (discussed in detail below); the original method is also sometimes referred to as Lehmer random number or the Park–Miller random number generator. They were introduced by Lehmer, and they represented a significant advancement [1, 6]. They generate sequences of random numbers by a linear equation, and their simplicity and speed made them popular, though they suffered from issues of periodicity and correlation as will be discussed below.

As mentioned above, it was clear from the very beginning that pseudorandom numbers have some specific problems. This became even more apparent in the 1960's and 1970's with the increasing use of PRNGs. This led to the development of rigorous statistical tests for randomness, such as the

χ^2-test, the Kolmogorov-Smirnov test, and the spectral test. These tests were, and remain, essential in evaluating and improving PRNG algorithms. During this era Donald Knuth's book "The Art of Computer Programming" [16] and George Marsaglia's [5] work on random number generation laid the foundation for modern PRNGs, providing comprehensive analyses and guidelines for generating pseudorandom numbers.

The Mersenne Twister by Matsumoto and Nishimura [12] in 1998, quickly became the standard in pseudorandom number generation due to its long period, high-dimensional equidistribution, and fast generation speeds. It remains widely used in applications and software, including programming languages like Python and R.

1.2.1 Cryptographically Secure Pseudorandom Number Generators

The 1970s onward saw an increased emphasis on cryptographic applications, necessitating PRNGs that could withstand cryptographic attacks. As a consequence, techniques such as Blum Blum Shub [9] and Fortuna [15] (the name stands for the Roman goddess of chance) were developed, focusing on cryptographic security over speed or statistical randomness, and finding use in secure communications and data encryption. New directions include methods such as Quantum Random Number Generators (QRNGs). QRNGs use quantum phenomena to generate numbers, promising a level of randomness that classical methods cannot achieve.

1.3 Requirements for a good Random Number Generator

Pseudorandom number generators (PRNGs) are fundamental tools in computational science, underpinning a vast array of applications from statistical modeling to cryptography. Understanding the requirements for effective PRNGs is crucial for researchers in various scientific and technical fields.

Fundamental Requirements of PRNGs:

1. Randomness, that is, uniform distribution: A key requirement for a PRNG is that it should produce numbers that are uniformly distributed over a specific range. This means each number within the range should have an equal probability of being generated.
2. Independence: Numbers generated should be statistically independent of each other. There should be no discernible relationship between successive numbers in the sequence. This can be analyzed, for example, using autocorrelation tests.
3. Long period. The PRNG should have a long period. This is especially important in simulations and cryptographic applications, to avoid repeating patterns. The Mersenne Twister has an extraordinarily long period of $2^{19937} - 1$.
4. High-Dimensional Uniformity: For complex simulations, especially in higher dimensions, the PRNG should maintain uniformity across all dimensions. This property can be examined using spectral tests.
5. The PRNG should be portable from one CPU architecture to another
6. The algorithm should be fast and use little memory
7. The algorithm should be parallelizable
8. For cryptography in particular:
    - Unpredictability and non-reproducibility: It is crucial that the output of the PRNG cannot be predicted or reproduced without knowledge of the initial state (seed). Fortuna and Yarrow are considered as Cryptographically Secure Pseudorandom Number Generators (CSPRNGs).
    - CSPRNGs should withstand attacks aimed at predicting future outputs or deducing

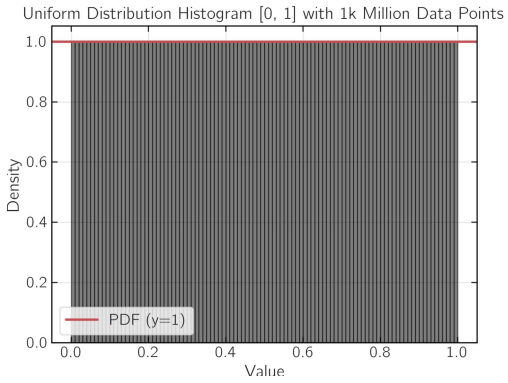

Figure 1.2: Uniform distribution in the interval $[0, 1]$. The histogram is sampled from 1,000,000,000 data points. The red line shows the exact result. The area under the histogram is equal to one. PDF = probability distribution function.

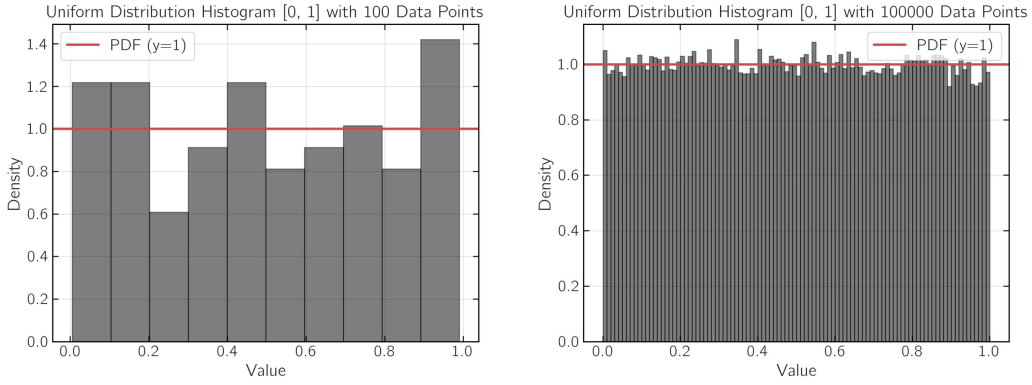

Figure 1.3: Uniform distribution in the interval $[0, 1]$. The histograms were sampled from 100 and 1000,000 data points. The red line shows the exact result. Note that the number of bins is different.

previous numbers from current ones.

1.4 Uniform Random Numbers

When talking about random numbers, we typically mean uniformly distributed random numbers. Typically we mean that random numbers are uniformly distributed between 0 and 1. It is then possible to generate other distributions from a uniform distribution, e.g., using the Box-Muller method. Uniform distribution is demonstrated in Figure 1.2.

It is important to note that, from the practical perspective, sampling enough data points is important. Figure 1.3 shows the dependence on the sample size. Creation of pseudorandom numbers is, however, not a trivial matter and many applications depends critically on it. In the following sections, we take a more detailed look at a few methods for random number generation.

1.4.1 Linear Congruential Generator (LCG)

Linear Congruential Generators are one of the oldest and most common types of PRNGs. As discussed above, they were introduced by Lehmer already in 1951 [1, 6] and they are based on, as the name suggests, a linear equation.

They are very easy to implement, they are fast, and use minimal amount of memory. They are also very portable, but the downside is that they do not produce high-quality random numbers. The problems include 1) correlations between the numbers, that is, successive numbers in the sequence are not independent, leading to potential patterns. 2) Limited period: The period of LCGs depends on the choice of parameters, and 3) LCGs can suffer from non-uniform distributions, especially in higher dimensions. Thus, when randomness is critical, LCGs should not be used. Such applications include calculation of critical exponents in continuous phase transitions and cryptography.

LCG generators are defined by the equation

$$x_{i+1} = (ax_i + c) \bmod m \tag{1.1}$$

and the parameters are defined as

- a: multiplier, $1 < a < m$.
- c: increment, $0 \leq c < m$.
- m: modulus (determines the period). The operation is even faster if m is chosen to be a power of 2.

Equation 1.1 returns a number in the interval $[0, m)$. To obtain a random number in the interval $[0, 1)$, we must scale by m. In addition, one needs a seed (x_0) to start the random number generator. The seed has to be supplied when the generator is called the first time. Using the above parameters, the following notation is typically used for LCGs:

$$\mathrm{LCG}(a, c, m). \tag{1.2}$$

Importantly, Equation 1.1 is a *recurrence relation* and numbers $\{x_i\}$ form a set of pseudorandom numbers. Notice also that the above equation defines a line with a modulo operation added. The modulo operation means that the result is wrapped if it exceeds the limit set by m since the modulo operation returns the remainder of $(ax_i + c)/m$.

An LCG generator is said to be of *full period* if the period is exactly m. The conditions for that are set by the Hull–Dobell Theorem [4] which states that for an LCG to have a period equal to the modulus m (full period), the following conditions must be satisfied:

Theorem 1.4.1 — The Hull-Dobell theorem.

1. Condition on c: The increment c must be *relatively prime* (or *coprime*) to the modulus m[a]. This means that c and m share no common factors other than 1.
2. Condition on a: $a - 1$ is divisible by all prime factors of m.
3. Condition for even modulus: If m is a multiple of 4, then $a - 1$ should be divisible by 4.

[a] The term "relatively prime" refers to a pair of integers that have no common positive divisor other than 1.

The Hull-Dobell Theorem is significant because it provides an easy-to-check criteria to ensure that an LCG has the longest possible period. This is crucial for the effectiveness of the LCG in generating pseudorandom numbers. If an LCG does not have a full period, it may repeat values too quickly (=has a short period) or exhibit patterns that could be predictable, reducing its effectiveness

It has been well established that LCGs are very sensitive to the choice of a, c, and m. Using a larger value for m (for a 32-bit computer one typically has $m = 2^{32}$) gives a long period as it goes through all accessible numbers. That is not enough, however, but a and c have to be chosen carefully as well. That is somewhat of an art and there are tables listing 'safe' numbers a, c, and m [13, 17]. LCGs have also the widely known property that the rightmost digits contain correlations. See Figure 1.4 for LCGs' sensitivity to parameter choices.

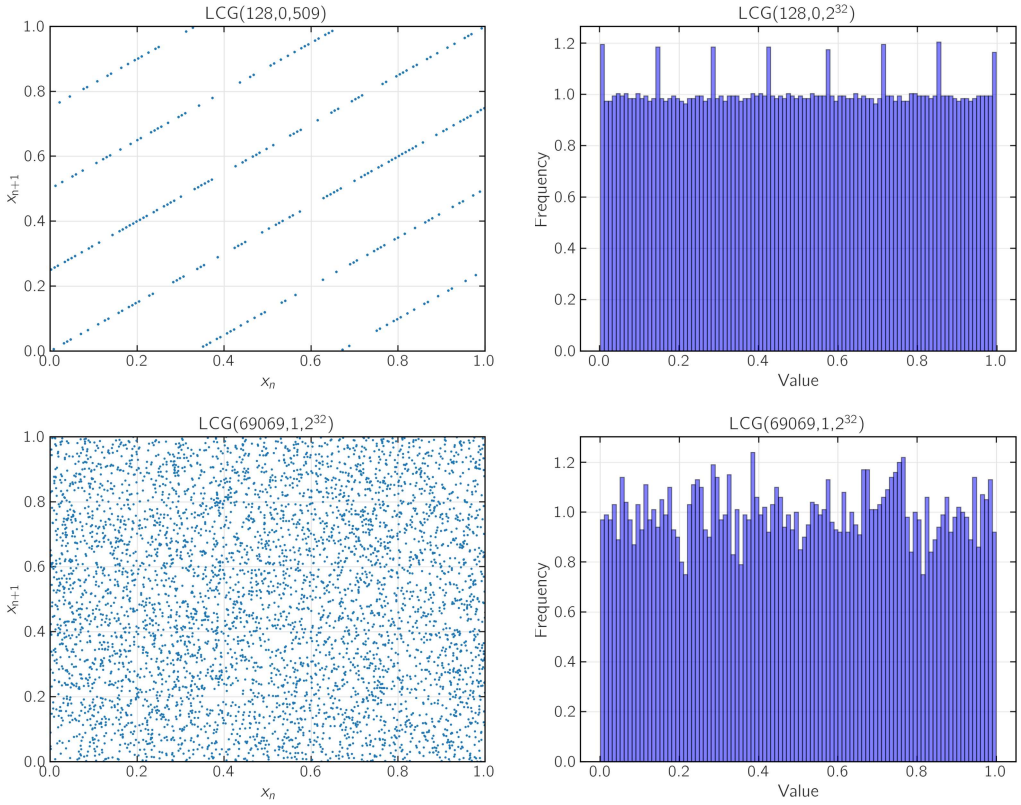

Figure 1.4: Demonstrations of poorly and well-chosen parameters for an LCG. The top right figure demonstrates hyperplanes and the corresponding histogram shows a regular pattern. The figures in the bottom row show much better behaviour.

Simple Python implementation of an LCG

```
def linear_congruential_generator(seed, a, c, m, n):
    x = seed
    numbers = []
    for _ in range(n):
        x = (a * x + c) % m
        numbers.append(x)
    return numbers

# Example parameters
seed = 42
a = 1664525
c = 1013904223
m = 2**32
n = 10

# Generate numbers
lcg_numbers = linear_congruential_generator(seed, a, c, m, n)
print(lcg_numbers)
```

Multiplicative Congruential Generators (MCG)

Generators with $c = 0$ in Equation 1.1 are called multiplicative congruential generators (MCGs); note that sometimes generators with $c > 0$ are called mixed. The quality of an MCG largely depends on the choice of a and m. The maximum period of an MCG is $m-1$, achievable under certain conditions, such as a and m being coprime[1]. For MCGs, the typical notation is thus

$$\mathrm{LCG}(a,m). \tag{1.3}$$

1.4.2 Fibonacci Sequence and the Fibonacci Method for Generating Random Numbers

Fibonacci Sequence

The Fibonacci sequence is defined as follows:

$$x_n = \begin{cases} 0 & \text{for } n = 0 \\ 1 & \text{for } n = 1 \\ x_{n-1} + x_{n-2} & \text{for } n > 1, \end{cases} \tag{1.4}$$

that is, the first two terms are zero and one ($x_0 = 1$ and $x_1 = 1$), and the rest of the terms are given as a sum of the two previous terms ($x_n = x_{n-1} + x_{n-2}$ for $n > 1$). With the above, the Fibonacci sequence up to $n = 10$ looks like this:

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
0	1	1	2	3	5	8	13	21	34	55

This sequence was know to Indian mathematicians even earlier, but Fibonacci introduced it in Europe in 1202.

Fibonacci Method for Generating Random Numbers

LCGs can be improved by adding more multipliers, i.e., we obtain the form

$$x_i = (a_1 x_{i-1} + \cdots a_r x_{i-r}) \bmod m. \tag{1.5}$$

Here we must have $r > 1$ and $a_r \neq 0$. If we choose $r = 2$, $a_1 = 1$ and $a_2 = 1$, we obtain the so-called Fibonacci generator

$$x_i = (x_{i-1} + x_{i-2}) \bmod m.$$

[1] Two integers are said to be coprime if the only positive integer (factor) that divides both of them is 1. In other words, coprime numbers do not have any common prime factors. Note that being coprime is an important concept in number theory and is used in various fields such as cryptography, particularly in algorithms like RSA (Rivest–Shamir–Adleman, 1977), where the choice of two coprime numbers is crucial for the algorithm's security.

Given the two 'seeds', each number is a sum of the two immediately preceding numbers. As such, this method for random number generation does not have particularly good properties. LCGs are typically preferred over them. The Fibonacci generator can be improved, however, and that will be discussed next.

1.4.3 Lagged Fibonacci Generator (LFG)

LFGs are based on a generalization of the above Fibonacci sequence, but with a twist: LFGs use *lagged terms* in the sequence to generate new numbers. Unlike the Fibonacci sequence, where each number is the sum of the two preceding numbers, the LFG allows for greater lags, often denoted as q and r, to generate the next term,

$$x_i = (x_{i-q} \otimes x_{i-r}) \bmod m. \tag{1.6}$$

Here, m is again the modulus, $q > r$ and the operation $\otimes$ is one of the following binary operations:

$+$ addition
$-$ subtraction
$\times$ multiplication
$\oplus$ XOR (exclusive OR) if m is a power of two.

Depending on the choice of the operation ($\otimes$), LFGs can be categorized into different types, such as additive, subtractive, multiplicative, or XOR-based[2] LFGs as indicated above. Notation for LFG generators is then the following:

$$\mathrm{LFG}(q, r, \otimes).$$

The quality of the output significantly depends on the choice of initial seed values and poor choices can lead to short periods and non-random behavior. Initialization of an LFG generator requires q seeds which can be produced using another RNG. LFGs can have a long period [16] and they seem to have good properties, but they have not been studied to such extent as the LCGs have. An example of a fairly commonly used LFG generator is RAN3 from Numerical Recipes [14]. The exact form of RAN3 is LFG(55,24,-).

Simple implementation of an LFG in Python

```
def lagged_fibonacci_generator(seed, j, k, op, m, n):
    if len(seed) < k:
        raise ValueError("Seed list must have at least k elements.")

    x = seed.copy()
    for _ in range(n):
        new_value = op(x[-j], x[-k]) % m
        x.append(new_value)

    return x[k:]

# Example usage
seed = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]  # Initial seed values
j, k = 7, 10  # Lags
op = lambda a, b: a + b  # Operation (addition)
m = 2**31  # Modulus
n = 10  # Number of values to generate

generated_sequence = lagged_fibonacci_generator(seed, j, k, op, m, n)
print(generated_sequence)
```

[2] Appendix A contains more information about logic gates.

1.4.4 Linear Feedback Shift Register (LFSR)

An LFSR is a shift register (a sequence of cells storing bits) where the input bit is a linear function of its previous state. The most commonly used linear function is the exclusive-or (XOR) operation. An LFSR generates a sequence of bits (0s and 1s). At each step, it shifts all bits to the right, and the leftmost bit (new bit) is calculated as the XOR of specific bits in the previous state, known as the tap positions. This linear feedback mechanism is what gives the LFSR its name. LFSRs are widely used in fields such as telecommunications for scrambling and descrambling signals, in cryptography for stream ciphers, and in digital circuit testing.

Characteristics of LFSRs include:

- Simplicity, speed, and ease of implementation. Capable of producing long, seemingly random sequences with appropriate tap settings.
- Vulnerable to cryptanalysis if used in isolation for cryptographic purposes. The sequence is ultimately deterministic and periodic. To enhance security for cryptographic applications, non-linear functions can be used for feedback.
- Multiple LFSRs can be combined to increase complexity and period.

Simple LFSR implementation in Python

```
def linear_feedback_shift_register(seed, taps, length):
    state = seed
    output = []
    for _ in range(length):
        next_bit = sum(state[tap] for tap in taps) % 2
        output.append(state.pop(0))  # Shift right
        state.append(next_bit)       # Append new bit

    return output

# Example parameters
seed = [1, 0, 0, 1, 1]  # Initial state
taps = [0, 2]           # Tap positions
length = 15             # Length of the sequence

lfsr_sequence = linear_feedback_shift_register(seed, taps, length)
print(lfsr_sequence)
```

1.4.5 Mersenne Twister

The Mersenne Twister [12] has emerged as one of the most widely used PRNGs in scientific and engineering applications, and it is currently the benchmark in pseudorandom number generation, balancing efficiency with statistical robustness. While not suitable for cryptographic purposes, its application in scientific simulations and analysis is unparalleled.

The Mersenne Twister is named for its use of Mersenne primes[3], that is, primes of the form

$$2^n - 1. \tag{1.7}$$

The most common version, MT19937, uses 2^{19937} ensuring a very long period of $2^{19937} - 1$ and high-dimensional equidistribution. Algorithmically, Mersenne Twister operates on a vector of n words and generates pseudorandom numbers by updating this state vector with bitwise operations and shifts. The state transition and output transformation functions are carefully designed to achieve fast computation and high-quality randomness.

Characteristics of the Mersenne Twister PRNG include

[3] A Mersenne prime is a prime number that can be expressed in the form $2^p - 1$, where p itself is a prime number. These primes are named after Marin Mersenne, a French mathematician from the 17$^{\text{th}}$ century who studied these numbers. The first few Mersenne primes are 3, 7, 31, and 127, corresponding to p values of 2, 3, 5, and 7, respectively.

- One of the longest periods achievable among known PRNGs.
- It achieves k-dimensional equidistribution for up to $k = 623$ dimensions, making it suitable for simulations requiring high-dimensional randomness.
- Efficiency: Provides a good balance between computational efficiency and quality of randomness.
- Not suitable for cryptographic applications due to predictability when a sufficient number of outputs are observed. Initialization can be relatively slow.

Python's `random` module uses Mersenne Twister:

```
import random

random.seed(123)  # Seeding the generator
numbers = [random.random() for _ in range(10)]  # Generating 10 random numbers
print(numbers)
```

1.4.6 Few words on implementation

Although the above methods look simple, one has to pay attention to their implementation. Vattulainen [10, 11], has pointed out that using single precision instead of double precision in the case of an LFG can lead to a decrease in period by a factor of about 10^7.

1.5 How to produce different distributions

So far we have only discussed uniformly distributed random numbers. There is a good reason for this - random numbers to produce any other distributions are almost always generated starting from random numbers distributed uniformly between 0 and 1. The challenge lies in converting this uniform distribution into other desired distributions.

Data, however, often comes in many other forms than uniformly distributed. For example, the peaks in various spectra have a Gaussian or Lorentzian shape, the decay of a radioactive sample follows an exponential function, and so on. Modeling in fields like finance, biology, and physics typically requires the use of different distributions. Different distributions are also needed with Monte Carlo methods to perform risk assessment, option pricing, or to solve complex integrals in physics, and in Machine Learning different distributions are needed for data generation for algorithm training and testing, where specific distributions may represent different data characteristics.

The practical question is: How to produce random numbers following a pre-defined distribution using random numbers which are evenly distributed in $(0,1)$. Why should evenly distributed random numbers be used as a starting point?

There are two basic approaches to generate other distributions than the uniform. The first is the analytical one, the other the numerical von Neumann rejection method. The rejection method involves generating points in a two-dimensional space and accepting or rejecting them based on a criterion related to the desired distribution: Generate two uniform random numbers, use one for the x-value and compute a test value (height) against the probability density function (PDF) of the target distribution.

1.5.1 Gaussian distribution: The Box-Muller algorithm

The Box-Muller method uses the Box-Muller transform to generate normally distributed random numbers [3]; the name Box-Muller refers to the British statistician George Box and American computer scientist/mathematician Mervin Muller. The method uses two uniformly generated

Technical Point 1.5.1 *Normal vs the Gaussian distribution & the Bell curve*

These are all the same thing. Physicists and chemists typically use the name *Gaussian distribution*, mathematicians prefer *normal distribution* and in social science and economics the term *Bell curve* is preferred.

random numbers ($\xi_1 \in (0,1)$ and $\xi_2 \in (0,1)$) as the source, and applies *the Box-Muller transform* for obtaining *a pair* of independent normally distributed random variates, that is, with their mean (μ) and variance (σ^2) defined as

$$\begin{cases} \mu & = 0 \\ \sigma^2 & = 1. \end{cases} \tag{1.8}$$

For normal distribution the mean, mode and median have the same value. The Box-Muller transform is based on the property that linear combinations of independent normally distributed random variables also follow a normal distribution.

In the Box-Muller method, two standard normal distributions in 2D are considered. Recall that the probability density function for a normal distribution is given as

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/(2\sigma^2)}. \tag{1.9}$$

If we now take two independent normal distributions, their joint distribution is given as

$$f(x,y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}y^2} = \frac{1}{2\pi} e^{-\frac{1}{2}(x^2+y^2)}. \tag{1.10}$$

Switching to polar coordinates, and remembering that the surface element transforms as $dxdy = rdrd\phi$ we get the polar density distribution function as

$$g(r,\phi) = \frac{1}{2\pi} e^{-\frac{1}{2}r^2}$$

Now, consider starting with two independent random variables, ξ_1 and ξ_2, which are uniformly distributed over the interval $(0,1)$. We use polar coordinates because they provide a convenient way to apply the properties of the uniform distribution. Consider two variables

$$R^2 = -2\ln(\xi_1) \tag{1.11}$$

$$\theta = 2\pi\xi_2. \tag{1.12}$$

Here, R^2 and θ were chosen since R^2 follows a χ^2 distribution with two degrees of freedom (which is an exponential distribution). The random variable θ is uniformly distributed in the interval $(0, 2\pi)$.

Now, we use the properties of the exponential and uniform distributions to construct the normal variables. We define g_1 and g_2 as

$$g_1 = \sqrt{-2\ln(\xi_1)}\cos(2\pi\xi_2) \tag{1.13}$$

$$g_2 = \sqrt{-2\ln(\xi_1)}\sin(2\pi\xi_2). \tag{1.14}$$

The new variables g_1 and g_2 are constructed to be jointly normal because they are linear combinations of R and θ, which have the necessary distributional properties. The new variables are standard normal because the transformation maps the uniform variables into a normal distribution with a

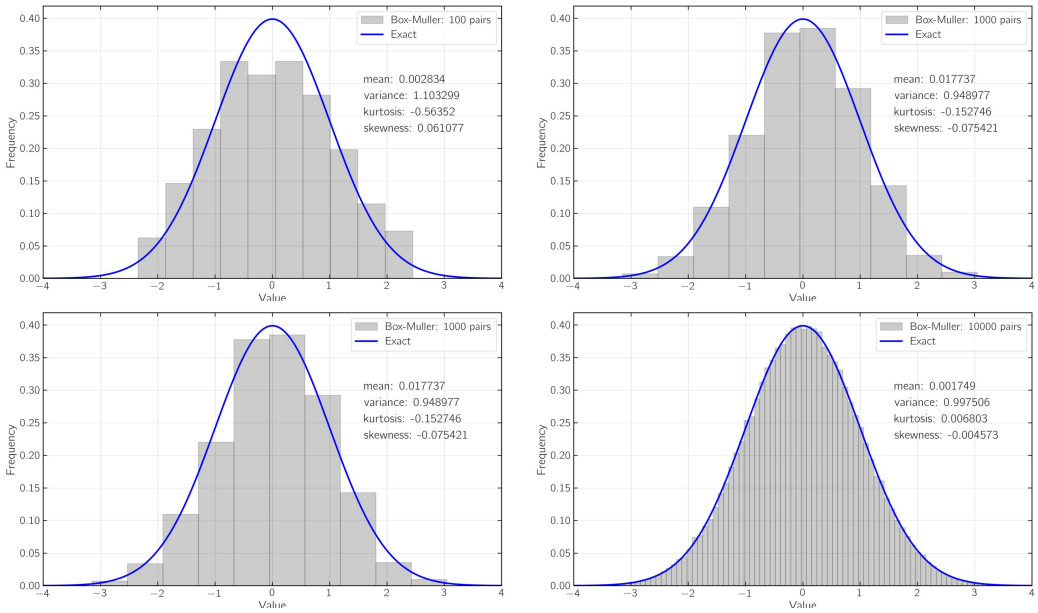

Figure 1.5: Random deviates created with the Box-Muller algorithm using different numbers of pairs. The statistical indicators are also shown in the figure. The solid line is the exact formula, Equation 1.9 with the properties given in Equation 1.8.

mean of 0 and a variance of 1. This can be shown using Jacobians to transform the probability density functions.

Algorithmically this method is fine, there is no wasted effort in terms of misses or something like that. But in terms of computational efficiency, it leaves much to be hoped for. It requires computation of the functions such a square root, log, sin and cos. Calculating all of these, and especially the last three ones, is excruciatingly slow compared to the simple arithmetic operations - it would be nice to have a more efficient routine.

The Box-Muller algorithm is crucial for simulations where the normal distribution plays a significant role, such as in finance for modeling stock prices, or in physics and chemistry for modeling particles and molecules behavior. Since many natural phenomena follow a normal distribution, the Box-Muller transform is widely applicable in various scientific research areas.

```
import math
import random

def box_muller_transform():
    U1, U2 = random.random(), random.random()
    Z0 = math.sqrt(-2 * math.log(U1)) * math.cos(2 * math.pi * U2)
    Z1 = math.sqrt(-2 * math.log(U1)) * math.sin(2 * math.pi * U2)
    return Z0, Z1

# Generate two standard normally distributed numbers
normal_variates = box_muller_transform()
print(normal_variates)
```

1.5.2 Lorentzian distribution

Lorentzian distribution, also known as the Cauchy distribution, is a probability distribution that is known for its "heavy tails," implying a higher probability of extreme values compared to the normal distribution, and it lacks defined mean and variance, making it an interesting subject in statistical analysis. It is often used in resonance phenomena and in describing distributions of particle sizes or

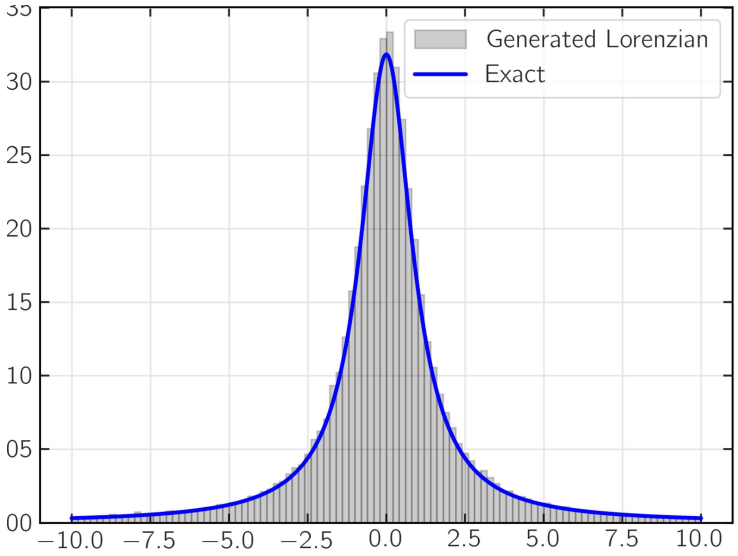

Figure 1.6: Lorentzian (Cauchy) distribution and comparison with the analytical form.

lifetimes in physics and chemistry. In financial modeling it is employed to model distributions with heavy tails, such as certain types of investment returns.

The Lorentzian (Cauchy) distribution is characterized by its probability density function (PDF):

$$f(x;x_0,\gamma) = \frac{\pi\gamma}{1+(\gamma(x-x_0))^2} \tag{1.15}$$

Lorentzian by Inverse Transform Sampling

Involves using the inverse of the cumulative distribution function (CDF) of the Lorentzian distribution. Method: Given a uniform random variable U, the Lorentzian variable X is obtained as:

$$X = x_0 + \gamma \tan\left[\pi\left(U - \frac{1}{2}\right)\right] \tag{1.16}$$

Python code:

```
import numpy as np
import matplotlib.pyplot as plt

def generate_lorenzian(x0, gamma, size=1000):
    U = np.random.uniform(0, 1, size)
    X = x0 + gamma * np.tan(np.pi * (U - 0.5))
    return X

# Parameters for Lorentzian distribution
x0, gamma = 0, 1

# Generate Lorentzian distributed numbers
data = generate_lorenzian(x0, gamma, 10000)

# Plotting the results
plt.hist(data, bins=100, density=True, range=[-10,10])
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Lorentzian Distributed Numbers')
plt.show()
```

Figure 1.6 shows the histogram and comparison with the analytical formula.

1.6 Problems

Problem 1.1 Equation 1.4 that defines the Fibonacci sequence is very simple. It is, however, a recursive and thus computationally expensive. Write a code that is a direct straightforward implementation of Equation 1.4. Determine the time complexity ($\mathcal{O}(n)$) of the function you wrote. Plot n vs t (t=time). Repeat the calculation at least 10 times to obtain error bars and include error bars in your plot. Remember to explain how your error bars were determined. What was the largest n you could compute within reasonable time (report the type of processor you are using).

Problem 1.2 Implementing different seeds: Write a simple python code that uses the current time as the seed for an LCG. Then, use the different seeds (and remember to record them in your plots!) and analyze the random number sequences generated. Do the sequences appear random and depend on the seed?

Problem 1.3 Parameter analysis: Experiment with different multipliers and moduli. How do the choices of a and m affect the sequence?

Problem 1.4 Statistical tests: Apply simple statistical tests (mean, variance) to the sequences. Do they conform to expectations for pseudorandom numbers?

Problem 1.5 Are the following pairs coprimes? a) 15 and 25, b) 8 and 15, c) 9 and 16

Problem 1.6 Consider an LCG with $m = 16$, $a = 5$, and $c = 3$. Check if it satisfies the Hull-Dobell Theorem.

References

[1]D. H. Lehmer, "Mathematical methods in large-scale computing units", Annu. Comput. Lab. Harvard Univ. **26**, 141–146 (1951).

[2]J. Von Neumann, "Various techniques used in connection with random digits", National Bureau of Standards, Applied Math. Series **12**, 36–38 (1951).

[3]G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates", aoms **29**, 610–611 (1958).

[4]T. E. Hull and A. R. Dobell, "Random number generators", SIAM Rev. **4**, 230–254 (1962).

[5]G. Marsaglia, "Random numbers fall mainly in the planes", Proc. Natl. Acad. Sci. U. S. A. **61**, 25–28 (1968).

[6]W. H. Payne, J. R. Rabung, and T. P. Bogyo, "Coding the lehmer pseudo-random number generator", Commun. ACM **12**, 85–86 (1969).

[7]W. A. Wagenaar, "Generation of random sequences by human subjects: a critical survey of literature", Psychol. Bull. **77**, 65–72 (1972).

[8]J. F. Monahan, "Extensions of von neumann's method for generating random variables", Math. Comput. **33**, 1065–1069 (1979).

[9]L. Blum, M. Blum, and M. Shub, "Comparison of two Pseudo-Random number generators", in Advances in cryptology (1983), pp. 61–78.

[10]I. Vattulainen, T. Ala-Nissila, and K. Kankaala, "Physical tests for random numbers in simulations", Phys. Rev. Lett. **73**, 2513–2516 (1994).

[11]I. Vattulainen and T. Ala-Nissila, "Mission impossible: find a random pseudorandom number generator", Computers in Physics **9**, 500 (1995).

[12]M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator", ACM Trans. Model. Comput. Simul. **8**, 3–30 (1998).

[13]P. L'Ecuyer, "Tables of linear congruential generators of different sizes and good lattice structure", Math. Comput. **68**, 249–261 (1999).

[14]Press, William H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C++: The art of scientific computing* (Cambridge University Press, Cambridge, Cambridgeshire, New York, 2002).

[15]N. Ferguson and B. Schneier, *Practical cryptography* (Wiley, Apr. 2003).

[16]D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms, Volume 2* (Addison-Wesley Professional, May 2014).

[17]G. Steele and S. Vigna, "Computationally easy, spectrally good multipliers for congruential pseudorandom number generators", (2020).

[18]M. Haahr, *RANDOM.ORG: true random number service*, (1998–2018) `https://www.random.org` (visited on 06/01/2018).