PART I

STOCHASTIC METHODS

Random Numbers 1

Random numbers (RNs) are important for scientific simulations. We will see that they are used in many different applications, including the following:

- To simulate random events in experimental data (e. g., radioactive decay)
- To simulate thermal fluctuations (e. g., Brownian motion)
- To incorporate a lack of detailed knowledge (e. g., traffic or stock market simulations)
- To test the stability of a system with respect to perturbations
- To perform random sampling

1.1 Definition of Random Numbers

Random numbers are a sequence of numbers in random order. The probability that a given number occurs next in such a sequence is always the same. Physical systems can produce random events, for example, in electronic circuits (*electronic flicker noise*) and in systems where quantum effects play an important role (e. g., radioactive decay or photon emissions in semiconductors). However, physical random numbers are usually *bad* in the sense that they are often correlated and not reproducible.

Generating random numbers with algorithms is also a bit problematic because computers are completely deterministic, while randomness should be nondeterministic. We therefore must content ourselves with generating so-called pseudo-random numbers, which are computed with deterministic algorithms based on strongly nonlinear functions. These numbers should follow a well-defined distribution and should have long periods. Furthermore, we should be able to compute them quickly and in a reproducible way.

A very important function for generating pseudo-random numbers is the modulo operator **mod** (% in C++). It determines the remainder of a division of one integer number by another one.

Given two numbers a (dividend) and n (divisor), we write a **modulo** n or a **mod** n, which represents the remainder of dividing a by n. For the mathematical definition of the modulo operator, we consider the integers a, q, and r. We then express a as

$$a = nq + r\,, \tag{1.1}$$

where r ($0 \leq r < |n|$) is the remainder (i.e., the result of a **mod** n). One useful property of the **mod** operator for generating RNs is that it is a strongly nonlinear function.

We distinguish between two classes of pseudo-random number generators (RNGs): multiplicative and additive RNGs.

- Multiplicative RNGs are simpler and faster to program and are based on integers.
- Additive RNGs are more difficult to implement and are based on binary variables.

In the following sections, we describe these RNGs in detail and outline different methods that allow us to examine the quality of random sequences.

1.2 Congruential RNG (Multiplicative)

The simplest form of a congruential RNG [2, 3] was proposed by Lehmer (see Figure 1.1). It is based on the **mod** operator.

Congruential RNG

To define the congruential RNG, we choose two integer numbers, c and p, and a seed value x_0 such that $c, p, x_0 \in \mathbb{Z}$. We then generate the sequence $x_i \in \mathbb{Z},\ i \in \mathbb{N}$ iteratively according to

$$x_i = (c x_{i-1}) \textbf{ mod } p\,. \tag{1.2}$$

Figure 1.1 Derrick H. Lehmer (1905–1991) was an American mathematician who worked on number theory and theory of computation.

This iteration generates random numbers in the interval $[0, p-1]$[1]. To transform a random number $x_i \in [0, p-1]$ into a normalized random number $z_i \in [0, 1)$, we simply divide x_i by p and obtain

$$0 \leq z_i = \frac{x_i}{p} < 1\,, \tag{1.3}$$

where z_i is a rational number (i.e., $z_i \in \mathbb{Q}$). The random numbers z_i are homogeneously distributed, which means that every number between 0 and 1 is equally probable.

Since all integers are smaller than p, the sequence must repeat after maximally $(p-1)$ iterations. Thus, the *maximal period* of the RNG defined by eq. (1.2) is $(p-1)$. If we pick the seed value $x_0 = 0$, the sequence remains at this value. Therefore, $x_0 = 0$ cannot be used as a seed of the described congruential RNG.

In 1910, the American mathematician Robert D. Carmichael [4] proved that the maximal period of a congruential RNG can be obtained if p is a Mersenne prime number and if it is the smallest integer number that satisfies

$$c^{p-1} \textbf{ mod } p = 1\,. \tag{1.4}$$

[1] Throughout the book, we adopt the notation that closed square brackets [ ] in intervals are equivalent to "≤" and "≥" and parentheses ( ) correspond to "<" and ">," respectively. Thus, the interval $[0, 1]$ corresponds to $0 \leq x \leq 1$, $x \in \mathbb{R}$ and $(0, 1)$ means $0 < x < 1$, $x \in \mathbb{R}$.

A *Mersenne number* is defined as $M_n = 2^n - 1$ with $n \in \mathbb{N}$. If the number M is prime, it is referred to as *Mersenne prime*. As of May 2021, only 51 Mersenne primes were known. The largest has 24,862,048 digits and was discovered by Patrick Laroche in December 2018 within the Great Internet Mersenne Prime Search (GIMPS) [5]. In 1988, Park and Miller [6] proposed the following numbers to generate the maximal period of congruential RNGs, here, in pseudo-code:

```
const int p=2147483647;

const int c=16807;

int rnd=42; // seed

rnd=(c*rnd)%p;

print rnd;
```

The number $p = 2147483648$ is a Mersenne prime and corresponds to the maximal integer of 32 bits: $2^{31} - 1$.

To assess the homogeneity of numbers generated by an RNG, we can either plot two consecutive pseudo-random numbers (x_i, x_{i+1}) (the so-called square test) or employ a "cube test" for three consecutive numbers (x_i, x_{i+1}, x_{i+2}). In Figure 1.2, we show an example of an RNG that clearly fails the cube test. All numbers that are generated by this particular RNG lie on just 15 parallel planes, indicating correlation effects that may lead to undesired effects in simulations of stochastic processes. In 1968, George Marsaglia (see Figure 1.3) showed that all congruential RNGs as defined in eq. (1.2) produce pseudo-random numbers that lie on equally spaced hyperplanes [8]. The distance between these planes decreases with the length of the period. For the interested reader, we briefly summarize the corresponding theorem on the following page.

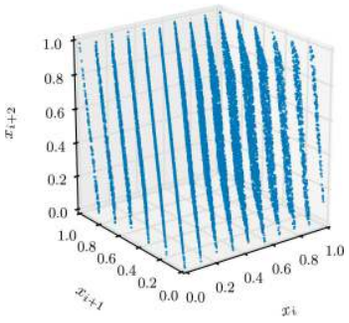

Figure 1.2 Cube test: plot of consecutive random numbers x_i, x_{i+1}, x_{i+2} with 15 clearly visible planes ("RANDU" algorithm with $c = 65539$, $p = 2^{31}$, $x_0 = 1$).

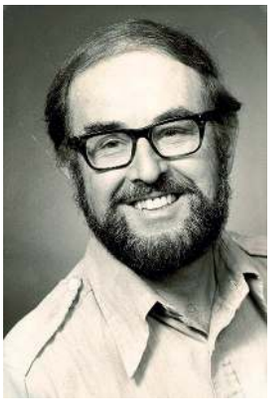

Figure 1.3 George Marsaglia (1924–2011) made many contributions to improving and testing RNGs [7].

Additional Information: Marsaglia's Theorem

Let $\{z_i\} = \{x_i/p\}$ $(i \in \mathbb{N})$ denote normalized numbers of an RNG sequence, and let $\pi_1 = (z_1, \ldots, z_n)$, $\pi_2 = (z_2, \ldots, z_{n+1})$, $\pi_3 = (z_3, \ldots, z_{n+2})$, ... be the points in the unit n-cube formed by n successive numbers z_i. Marsaglia showed that all points $\pi_1, \pi_2, \ldots$ lie on parallel hyperplanes [8]. Formally, if $a_1, a_2, \ldots, a_n \in \mathbb{Z}$ is any choice of integers such that

$$a_1 + a_2 c + a_3 c^2 + \cdots + a_n c^{n-1} \equiv 0 \,\textbf{mod}\, p,$$

then all the points $\pi_1, \pi_2, \ldots$ will lie in the set of parallel hyperplanes defined by the equations

$$a_1 y_1 + a_2 y_2 + \cdots + a_n y_n = 0, \pm 1, \pm 2, \ldots, \quad y_i \in \mathbb{R},\ 1 \le i \le n.$$

Additional Information: Marsaglia's Theorem (*cont.*)

There are at most

$$|a_1| + |a_2| + \cdots + |a_n|$$

of these hyperplanes that intersect the unit n-cube. Note that there is always a choice of $a_1, \ldots, a_n$ such that all of the points fall in fewer than $(n!p)^{1/n}$ hyperplanes. The theorem can be proven in four steps:
Step 1: If

$$a_1 + a_2 c + a_3 c^2 + \cdots + a_n c^{n-1} \equiv 0 \,\mathbf{mod\ p},$$

one can prove that

$$a_1 z_i + a_2 z_{i+1} + \cdots + a_n z_{i+n-1}$$

is an integer for every i.
Step 2: The point $\pi_i = (z_i, z_{i+1}, \ldots, z_{i+n-1})$ must lie in one of the hyperplanes

$$a_1 y_1 + a_2 y_2 + \cdots + a_n y_n = 0, \pm 1, \pm 2, \ldots, \quad y_i \in \mathbb{R},\ 1 \leq i \leq n.$$

Step 3: The number of hyperplanes of the above type, which intersect the unit n-cube, is at most

$$|a_1| + |a_2| + \cdots + |a_n|\,.$$

Step 4: For every multiplier c and modulus p, there is a set of integers $a_1, \ldots, a_n$ (not all zero) such that

$$a_1 + a_2 c + a_3 c^2 + \cdots + a_n c^{n-1} \equiv 0 \,\mathbf{mod\ p}$$

and

$$|a_1| + |a_2| + \cdots + |a_n| \leq (n!p)^{1/n}\,.$$

This is of course only the outline of the proof. The exact details are described in Marsaglia (1968) [8]. In a similar way, it is possible to show that for congruential RNGs the distance between the planes must be larger than $\sqrt{\frac{p}{n}}$.

Exercise: Congruential RNG

Task 1
Write a program that generates random numbers using a congruential random number generator. First, use small values for c (< 50) and p (< 50), for example, $c = 3$ and $p = 31$.

- Check for correlations using the square test. That is, create a plot of two consecutive random numbers (x_i, x_{i+1}). (What is the maximum number of random numbers that have to be created until you can see all possible lines/planes for this specific random number generator?)
- Create a corresponding 3D plot for the cube test.
- Do the same for other random number generators (at least one!), for example, by changing c and p. You may also compare your results to those obtained with C++ built-in generators, such as `rand()` and `drand48()`.

Exercise: Congruential RNG (*cont.*)

Task 2
Generate a homogeneous distribution of random points inside a circle. How should the coordinates r and ϕ be chosen using uniformly distributed random numbers?

Task 3
Test your RNG for different c and p using the χ^2 test:

- Divide the range of random numbers into k bins. That is, divide the range into discrete intervals of the same size, so that the probability of a random number lying in interval i is $p_i = 1/k$.
- Using each RNG, generate at least one sequence of n numbers. For each sequence, determine N_i, the number of random numbers in interval i (choose n such that all $np_i \geq 5$).
- Compute the χ^2 value for one specific sequence s of random numbers

$$\chi^2_s = \sum_{i=1}^{k} \frac{(N_i - np_i)^2}{np_i} .$$

  Use the table from Knuth [9] to check if the random numbers are uniformly distributed and compare the quality of different RNGs.
- Calculate χ^2_s for different sequences (i. e., different seeds of the RNG). You can then plot the cumulative probability for the χ^2 in comparison to the theoretically expected values (values again from the table of Ref. [9]).

1.3 Lagged Fibonacci RNG (Additive)

A slightly more complicated RNG is the lagged Fibonacci algorithm proposed by Robert C. Tausworthe (see Figure 1.4) in 1965 [10]. Lagged Fibonacci-type generators can achieve extremely long periods and allow us to make some predictions about their underlying correlation effects.

To define lagged Fibonacci RNGs, we start with a sequence of binary numbers $x_i \in \{0, 1\}$ $(1 \leq i \leq b)$. The next bit in our sequence x_{b+1} is

$$x_{b+1} = \left(\sum_{j \in \mathcal{J}} x_{b+1-j} \right) \mathbf{mod}\, 2 \,, \tag{1.5}$$

where $\mathcal{J} \subset [1, \ldots, b]$. The sum $\sum_{j\in\mathcal{J}} x_{b+1-j}$ includes only a subset of all the other bits, so the new bit could, for instance, simply be based on the first and third bits, $x_{b+1} = (x_1 + x_3)\mathbf{mod}\, 2$ (or any other subset). We now focus on the properties of RNGs that are defined by eq. (1.5) and consider a two-element lagged Fibonacci generator.

Figure 1.4 Robert C. Tausworthe is a retired senior research engineer at the Jet Propulsion Laboratory, California Institute of Technology.

Two-Element Lagged Fibonacci RNG

Let $c, d \in \{1, \ldots, b\}$ with $d \leq c$. The RNG sequence elements x_{b+1} are recursively generated according to

$$x_{b+1} = (x_{b+1-c} + x_{b+1-d}) \textbf{ mod } 2 .$$

We immediately see that we need some initial sequence of at least c bits to start from (a so-called *seed sequence*). Except for all bits equal to zero, any other seed configuration can be used. One possibility is to use a seed sequence that was generated by a congruential RNG.

The maximum period of sequences that are generated by the outlined two-element lagged Fibonacci RNG is $2^c - 1$. As for congruential RNGs, there are conditions for the choice of the parameters c and d to obtain the maximum period. For lagged Fibonacci generators, c and d must satisfy the Zierler–Trinomial condition, which states that the trinomial

$$T_{c,d}(z) = 1 + z^c + z^d , \tag{1.6}$$

where z is a binary number, cannot be factorized into subpolynomials (i. e., is primitive). A possible choice of numbers that satisfy the Zierler condition is $(c, d) = (250, 103)$. The generator is named after Kirkpatrick (see Figure 9.10) and Stoll, who proposed these numbers in 1981 [11].

Some examples of known pairs (c, d) follow:

$$\begin{array}{c} (c,d) \\ (250, 103) \quad \text{Kirkpatrick–Stoll (1981) [11]} \\ (132049, 54454) \quad \text{J. R. Heringa et al. (1992) [12]} \\ (6972593, 3037958) \quad \text{R. P. Brent et al. (2003) [13].} \end{array} \tag{1.7}$$

We may use one of the following methods to convert the obtained binary sequences to natural numbers (e. g., 32 bit unsigned integers):

- Running 32 lagged Fibonacci generators in parallel (this can be done very efficiently): The problem with this method is the initialization, because all 32 initial sequences have to be uncorrelated. The quality of the initial sequences has a major impact on the quality of the produced random numbers.
- Extracting a 32-bit-long part from the sequence: This method is relatively slow because for each random number we have to generate 32 new elements in the binary sequence.

1.4 Available Libraries

In general, we should always make sure that we use a high-quality RNG. For some purposes it might be sufficient to use `drand48` (in C/C++). If your compiler supports

the C++11 standard (or above), there are different implementations already available in the "random" library. For example, linear congruential RNGs are available by calling `minstd_rand` from the `linear_congruential_engine` class. A useful general-purpose RNG is the so-called *Mersenne twister* [14], which was developed in 1997 by Makoto Matsumoto and Takuji Nishimura. The Mersenne twister has some structural similarities to lagged Fibonacci RNGs and belongs to the class of generalized feedback shift register algorithms. The name "Mersenne twister" was chosen because its period length is a Mersenne prime. In C++, we can use the Mersenne twister to generate uniformly distributed random numbers as follows:

```
#include <iostream>
#include <random>

using namespace std;
int main()
{

    random_device rd;
    mt19937 mt_rng(rd());
    uniform_real_distribution<double> u(0.0, 1.0);

    double rnd = u(mt_rng);

    cout << rnd << endl;

    return 0;
}
```

In the above code listing, we use the `random_device` as seed for the `mt19937` generator. This generator has a very long period of $2^{19937} - 1$. For the sake of reproducibility of numerical results, it is recommended to store all seeds in a file. In PYTHON, no further efforts are required to use the Mersenne twister because the `mt19937` is the core RNG in all PYTHON distributions.

1.5 How Good is an RNG?

There are many possibilities to test how "random" a certain RNG sequence is. Possible tests for a given sequence $\{s_i\}$, $i \in \mathbb{N}$ include the following:

1. Square test (see Section 1.2 for details)
2. Cube test (see Section 1.2 for details)

3. Average value: the arithmetic mean of all numbers in the sequence $\{s_i\}$ should correspond to the analytical mean value. Let us assume here that the numbers s_i are rescaled to lie in the interval $s_i \in [0, 1)$. The arithmetic mean should then be

$$\bar{s} = \lim_{N\to\infty} \frac{1}{N} \sum_{i=1}^{N} s_i = \frac{1}{2}\,. \tag{1.8}$$

   The more numbers that are averaged, the better $\frac{1}{2}$ will be approximated.
4. Fluctuation of the mean value (χ^2 test): the distribution around the mean value should behave like a Gaussian distribution.
5. Spectral analysis (Fourier analysis): Let $\{s_i\}$ denote values of a function. It is possible to perform a Fourier transform of such a function by means of the fast Fourier transform (FFT; see details in Section 15.2.1). If the frequency distribution corresponds to white noise (uniform distribution), the randomness is good; otherwise, peaks will show up (resonances).
6. Correlation test: Analysis of correlations such as

$$\langle s_i s_{i+d}\rangle - \langle s_i^2\rangle, \tag{1.9}$$

   for different values of d.

Of course, this list is not complete. Many other tests can be used to check the quality of RNG sequences.

Probably the most famous set of RNG tests is the Marsaglia's "Diehard" set. These Diehard tests are a battery of statistical tests for measuring the quality of a set of random numbers. They were developed over many years and published for the first time by Marsaglia on a CD-ROM with random numbers in 1995 [15]. Marsaglia's tests were inspired by different applications, and each can measure different types of correlations.

Additional Information: Marsaglia's "Diehard" Tests

- Birthday spacings: If random points are chosen in a large interval, the spacings between points should be asymptotically Poisson distributed. The name stems from the birthday paradox.[1]
- Overlapping permutations: When analyzing five consecutive random numbers, the 120 possible orderings should occur with equal probability.
- Ranks of matrices: A number of bits of some number of random numbers is formed into a matrix over {0,1}. The rank of this matrix is then determined, and the ranks are counted.
- Monkey test: Sequences of some number of bits are taken as words, and the number of overlapping words in a stream is counted. The number of words not appearing should follow a known distribution. The name is based on the infinite monkey theorem.[2]
- Parking lot test: Randomly place unit circles in a 100×100 square. If the circle overlaps an existing one, try again. After 12,000 tries, the number of successfully "parked" circles should follow a certain normal distribution.

Additional Information: Marsaglia's "Diehard" Tests (*cont.*)

- Minimum distance test: Find the minimum distance of 8,000 uniformly randomly placed points in a $10{,}000 \times 10{,}000$ square. The square of this distance should be exponentially distributed with a certain mean.
- Random spheres test: Put 4,000 randomly chosen points in a cube of side length 1,000. Now a sphere is placed on every point with a radius corresponding to the minimum distance to another point. The smallest sphere's volume should then be exponentially distributed.
- Squeeze test: 2^{31} is multiplied by random floats in $[0, 1)$ until 1 is reached. After 100,000 repetitions, the number of floats needed to reach 1 should follow a certain distribution.
- Overlapping sums test: Sequences of 100 consecutive floats are summed up in a very long sequence of random floats in $[0, 1)$. The sums should be normally distributed.
- Runs test: Ascending and descending runs in a long sequence of random floats in $[0, 1)$ are counted. The counts should follow a certain distribution.
- Craps test: 200,000 games of craps[3] are played. The number of wins and the number of throws per game should follow a certain distribution.

[1] The birthday paradox states that the probability of two randomly chosen persons having the same birthday in a group of 23 (or more) people is more than 50 percent. For 57 or more people, the probability is already larger than 99 percent. Finally, for at least 366 people, the probability is exactly 100 percent. This is not paradoxical in a logical sense; it is called a paradox nevertheless since intuition would suggest probabilities much lower than 50 percent.
[2] The infinite monkey theorem states that a monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely (i.e., with probability 1) write a certain text, such as the complete works of William Shakespeare.
[3] A dice game.

1.6 Nonuniform Distributions

Thus far, we have only considered uniform distributions of pseudo-random numbers. The congruential and lagged Fibonacci RNGs produce numbers that can easily be mapped to the interval $[0, 1)$ or any other interval by simple shifts and multiplications. However, if we want to generate random numbers that are distributed according to a certain distribution (e. g., a Gaussian distribution), the algorithms presented so far are not able to do so. However, we may employ techniques that permit us to transform uniform pseudo-random numbers to other distributions. There are essentially two different ways to perform this transformation:

- We can apply *transformation methods* if the target distribution is known analytically, is integrable, and the resulting expression is invertible.
- However, if the target distribution is not known analytically or if it cannot be analytically integrated and inverted, we have to use the so-called *rejection method*.

These methods are explained in the following sections.