2. Probabilities, Distributions and Random Numbers

2.1 From Numbers to Random Numbers

A sequence of numbers that is random plays a very important role for the stochastic simulation methods and for the deterministic methods. For the stochastic simulation methods we need random numbers for example to decide whether a lattice site is considered occupied or un-occupied (c.f. previous chapter). For deterministic simulation methods we need to set the initial momenta of particles. Both examples show that random numbers are needed that are distributed according to a prescribed distribution. In the first example the distribution is uniform. Each random number has the same a priori probability. In the second example the momenta should obey a Boltzmann distribution.

In general, we need numbers $x \in [a, b]$ with a probability density $f(x)$ such that

$$\text{Prob}[x_1 < X < x_2] = \int_{x_1}^{x_2} f(x)dx \tag{2.1}$$

with the corresponding distribution

$$F(x) = \text{Prob}[X < x] = \int_{a}^{x} f(x')dx' \tag{2.2}$$

Our task is to device algorithms that produce numbers in the given interval that are in some sense random irrespective of the distribution.

We certainly can not use a "physical" random number generator. Consider testing a program where possible problems can not be located due to the non-reproducibility of the outcome of a run of the program under given initial conditions. Also, we need to be sure that the random numbers are generated at a constant and fast enough rate.

A random number generator should produce numbers that appear random, i.e., passes the common statistical tests for randomness and the sequence of numbers must be reproducible. The underlying algorithm will be deterministic, and we can only hope for "pseudo-random numbers". Since the random numbers come from a deterministic algorithm, we can always device a statistical test where the specific algorithm fails to appear random.

A more technical issue is the portability of the random number generator from one operating system to the another. Also we need to consider the portability from one processor type (for example from a 64-bit machine to a 128-bit machine) the another.

Intuitively, we can list a number of criteria that a sequence of numbers must fulfil to pass as a random number sequence:

- unpredictability,
- independence,
- without pattern.

These criteria appear to be the minimum request for an algorithm to produce random numbers. More precisely we can formulate:

- prescribed distribution,
- uncorrelated,
- passes every test of randomness,
- large period before the sequence repeats (see later),
- sequence repeatable and possibility to vary starting values,
- fast algorithm,
- portability.

The most important desired properties are, of course, the statistical properties. Unfortunately there is no generator available without fault, but the list of known properties is growing. This reduces the risk of applying a particular bad one in a simulation.

Computational efficiency is extremely important. Simulation programs may require huge numbers of random numbers, so the computation of a single random number must be very fast.

2.2 Distributions

A very simple test of the statistical properties of an empirical distribution is to compute the moments of the distribution. Here we apply this to generated random numbers. Assume that we generate n numbers. We divide the interval of the random numbers into bins and count the number of occurrences of numbers that belong to a given interval. This produces a histogram counting the number of observations for each of the bins (frequencies). Dividing all frequencies by the total number of observations n gives the probabilities for the bins. For the random numbers we want the distribution $F(x)$ to be the same for every interval $[x, x + dx]$. A distribution of random numbers where all numbers have the same probability is called a *uniform distribution*. We define the *first* and the *k-th moment* of a distribution as

$$< x > = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.3}$$

$$< (x- < x >)^k > = \frac{1}{n}\sum_{i=1}^{n}(x_i- < x >)^k \qquad k = 2, ... \tag{2.4}$$

Let us assume that the numbers x_i are normalized to the unit interval, then we must have 1/2 for the first moment or the *expectation value*. Because of the symmetry of the uniform distribution all odd moments must vanish. We can define the *skew of a distribution* of numbers as

$$\text{skew} = \frac{< (x- < x >)^3}{[< (x- < x >)^2 >]^{3/2}} \quad . \tag{2.5}$$

The skew reflects how far a distribution departs from being symmetric. Also of interest is the deviation from the *normal (standard or Gauss) distribution*. This is measured by the excess

$$\text{excess} = \frac{< (x- < x >)^4 >}{[(< (x- < x >)^2 >]^2} - 3 \quad . \tag{2.6}$$

This quantity will be useful for us later when we want to study phase transitions. There, the departure from the gaussian distribution signals the phase transition, and the location can be determined using the excess.

2.3 Generating Random Number Generators

The most common generators use very basic operations and apply them repeatedly on the numbers generated in previous steps. We formulate this as a recursion relation

$$x_{i+1} = G(x_i), \qquad x_0 = \text{initial value} \quad , \tag{2.7}$$

where we have made explicit only the dependence on the immediate predecessor. The most important representatives of this class of generators are the

- linear congruential,
- lagged Fibonacci,
- shift-register, or a
- combination of linear congruential.

For the moment we consider generator functions G, that produce numbers in a given interval, or a model set with a uniform distribution. A uniform distribution is one, where all possible numbers in a given interval occur with the same probability.

A very simple generator is constructed using the modulo function.

$$G(x) = (xa + b) \bmod M \tag{2.8}$$

This function produces a dilatation, translation and a folding back into the interval. Random number generators based on this function are called *linear congruential generators* or LCG(a,b,M) for short. If we assume integers as the set on which the modulo function is defined, then for example, the range of integer numbers for a 32-bit architecture is at most $M = 2^{31} - 1$. Here we assume that one bit is taken for the sign of the number. Then the numbers range at most from 0 to $M - 1$. Of course we can map these onto the real interval between 0 and 1, recognizing that this is an approximation to the real-valued random numbers.

The choice of the parameters a, b and M determine the statistical properties and how many different numbers we can expect, before the sequence repeats itself. The period can be shown to be maximal, if M is chosen to be a prime number. Then the whole range of numbers can occur.

Here we only consider modulo generators with $b = 0$. Such generators are called *multiplicative*, and the short form MLCG(a,M) is used for such generators. These are the most commonly used, since one can show that *additive* generators, i.e. generators with b zero have undesirable statistical properties.

The choices for the parameter a are manifold. For example $a = 16807$, 630360016 or 397204094 are possible choices with $M = 2^{31} - 1$. There are particular bad choices for the multiplier a. For example $a = 65539$ is such a bad choice. The decision, whether a particular value for a is a good or bad choice, with respect to the statistical properties, can be made based on theorems that limit the possible choices for a. Also some statistical tests exclude values for a. Here we do not have space to go into these details and the interested reader is directed to the literature.

In the following, we list some modulo generators that have reasonably, within the known limitations, good properties

Recursion Relation	Period
$x_{i+1} = 16807x_i \bmod (2^{31-1} - 1)$	$2^{31} - 1$
$x_{i+1} = 69069x_i + 1 \bmod (2^{32})$	2^{32}
$x_{i+1} = 1664525x_i + 1 \bmod (2^{32})$	2^{32}

The introduced generator is easy to implement on a computer. Using, in its straightforward form, the intrinsic modulo function, the algorithm reads

```
ix = MOD(a * ix, m)
```

The MOD function has the advantage of transportability of the program from one machine to another. Its disadvantage is the computational inefficiency. Many operations are carried out, among them the most time-consuming, the division.

Another type of generator used very frequently in simulations is the *lagged Fibonacci generator* . The lagged Fibonacci generator, symbolically denoted by LF(p,q,⊗) with $p > q$, is based on a Fibonacci sequence of numbers with respect to an operation which we have given the generic symbol ⊗. Let S be the model set for the operation ⊗, for example the positive real numbers, the positive integers, or the set $S = 0, 1$. The binary operation ⊗ computes a new number from previously generated numbers with a lag p

$$x_n = x_{n-p} \otimes x_{n-q} \quad , \quad p > q \qquad . \tag{2.9}$$

To start the generator we need p numbers. These can be generated using for example a modulo generator. The advantage of the lagged Fibonacci generator, apart from removing some of the deficiencies that are build into the modulo type generators, is that one can operate on the level of numbers or on the level of bits. For example, we can construct a *generalized shift-register generator* GFSR(p,q,⊗), where the operation is interpreted as the *exclusive or*, which acts on every of the 32 bits in a computer word. This generator is also known under the name of $R250$. The generator $R250$ is the generator used in many of the simulation programs accompanying this book.

Let us turn now to the generation of non-uniform distributions. First we look at the normal or Gaussian distribution. Sampling such a distribution is computationally rather inefficient compared to the algorithms described above. But in the applications described in this book, the generation of a Gaussian distribution is not time critical.

Typically algorithms generating non-uniform variates do so by converting uniform variates. In its most straightforward form a normal deviate x with mean $< x >$ and standard deviation σ is produced as follows [**?**, **?**].

Let n be an integer, determined by the needed accuracy. Then

1. sum n uniform random numbers r_i from the interval $(-1, 1)$:

$$s_n = \sum_{i=1}^{n} r_i$$

2. and let $x =< x > +\sigma s_n \sqrt{3.0/n}$

For some purposes the simple method will be sufficient, but if good accuracy is needed the above algorithm should be avoided. More efficient and accurate is the idea of von Neumann [**?**] with the modification of Forsythe [**?**].

Let $G(x)$ be a function on the interval $[a, b]$ with $0 < G(x) < 1$ and $f(x)$ the probability distribution $f(x) = a \exp[-G(x)]$, where a is a constant.

1. Generate r from a uniform distribution on $(0, 1)$
2. Set $x = a + (b - a)r$.
3. Calculate $t = G(x)$.

4. Generate $r_1, r_2, ..., r_k$ from a uniform distribution on $(0, 1)$ where k is determined from the condition

$$t > r_1 > r_2 > ... > r_{k-1} < r_k$$

   If $t < r_1$, then $k = 1$.
5. If k is even, reject x and go to 1; otherwise, x is a sample.

Again, the method converts uniform random numbers into non-uniform ones.

An interesting method for generating normal variates is the polar method [**?**]. It has the advantage that two independent, normally distributed variates are produced with practically no additional cost in computer time.

1. Generate two independent random variables, U_1, U_2 from the interval $(0, 1)$.
2. Set $V_1 = 2U_1 - 1$, $V_2 = 2U_2 - 1$
3. Compute $S = V_1^2 + V_2^2$.
4. If $S \geq 1$, return to step 1.
5. Otherwise, set

$$x_1 = V_1\sqrt{-2\ln S/S} \tag{2.10}$$

$$x_2 = V_2\sqrt{-2\ln S/S} \tag{2.11}$$

2.4 Testing Random Numbers

A number of statistical tests have been devised to check for the properties of random number generators. To name just a few prominent tests

- χ^2,
- Kolmogorov-Smirnov,
- correlation,
- run and
- visual test.

The statistical tests are tests how well the empirical distribution, i.e., the generated sequence, fits a test distribution. Of these, we will not describe any and direct the reader to the literature. We shall discuss, however, the run and the lattice test.

The *run test* tests whether an empirical distribution has monotone decreasing or increasing subsequences and confronts these with the expectation for their occurrence. Let us assume that we want to test for monotone increasing subsequences. Such a test is called a *run test up*, otherwise *run test down*. A run of length r of a sequence $x = (x_1, ..., x_n)$ is a maximal strictly monotonically increasing (decreasing) subsequence $(x_i, ..., x_{i+r-1})$, i.e.,

$$x_{i-1} > x_i < ... < x_{i+r-1} > x_{i+r} \tag{2.12}$$

with x_0 positive infinite and x_{n+1} negative infinite.

The expectation for the distribution of runs is derived from a simple permutation argument and will not be reproduced here. For large n, one can show, that the probability to get a run of length r is given by $r/(r+1)!$, hence

$$1/2, 1/3, 1/8, 1/30, 1/144, ... \tag{2.13}$$

The way the test is derived shows that this tests for correlation in the generated sequence. The expected probabilities for the sequences reflect the independence from correlation.

A very easy test is *lattice test*. Suppose we have to visit the sites of a simple cubic lattice L^3 at random. The three coordinates are obtained from three successively generated random numbers $r_1, r_2, r_3 \in (0, 1)$, as

$$\begin{aligned} ix &= rz * L + 1 \\ iy &= rz * L + 1 \\ iz &= rz * L + 1 \end{aligned}$$

where ix, iy, iz are integer variables, implying a conversion of the real right-hand sides to integers, i.e., removal of the fractional part. If there are no correlations between successively generated random numbers all sites will eventually be visited. However, only certain hyperplanes are visited if correlations exist. This was most impressively demonstrated first by Lach [**?**].

2.5 Problems

1. Generate a sequence of random numbers with parameters you think are appropriate (not the ones listed above!). Visualize this sequence. Check whether the distribution is uniform.
2. Generate a sequence of random numbers with parameters you think are appropriate (not the ones listed above!). Store this sequence and compute the moments of the distribution. Check whether the distribution is uniform. Compare your findings with a graphical output (see for example the section on correlation).
3. For the shift-bit-register generator generate n sequences of length m. What will be the distribution of the expectation values?
4. Use the run test to test different random number generators. Compare your findings with a visual analysis and an analysis of the moments.
5. Can you construct a modulo generator which gives an expectation value of 1/2, but all other moments do not correspond to the uniform distribution?