Part I

Simulating Statistical Models

STATISTICAL MODELING

- PROBABILISTIC MODELING
  - → ways to describe uncertainty
  - → sampling methods]
- STATISTICAL INFERENCE
  1. likelihood methods
     - → optimization problems
  2. Bayesian methods
     - → integrals computation]

Monte Carlo methods

$X_1, \dots, X_n \sim f(x|\theta_1, \dots, \theta_k)$

likelihood:

$$L(\theta_1, \dots, \theta_k | x_1, \dots, x_n) = \prod_{i=1}^{n} f(x|\theta_1, \dots, \theta_k)$$

$\vec{\theta} = \arg\max L(\vec{\theta}|\vec{x})$ ← parameter estimation

(optimization problem)

Bayesian methods:

$X \sim f(x|\theta)$

plus prior information on θ

(now θ is a random variable itself characterized by a given distribution $\pi(\theta)$)

Posterior distribution of θ:

distribution of θ once we have seen the data: (Bayes theorem)

$$\pi(\theta|x) = \frac{f(x|\theta)\,\pi(\theta)}{\int f(x|\theta)\pi(\theta)\,d\theta}$$

$\propto$ likelihood · prior

(**Bayesian** context)

Suppose that we want to estimate some function of the parameter: $h(\theta)$.

We can introduce a loss:

$L(\delta, \theta)$ = error that we do by estimating $h(\theta)$ with δ

e.g.

$L(\delta, \theta) = \| h(\theta) - \delta \|^2$

→ we want to minimize the Bayesian risk, namely:

$$\iint \underbrace{L(\delta,\theta)\, f(x|\theta)\,\pi(\theta)}_{\propto \text{ posterior}}\, dx\, d\theta$$

→ we have to deal with integral computation

One of the topics of this course is the study of statistical models using computer simulations. Here we use the term *statistical models* to mean any mathematical models which include a random component. Our interest in this chapter is in *simulation* of the random component of these models. The basic building block of such simulations is the ability to generate random numbers on a computer (Chapters 1-2). Generation of random numbers, or more general random objects, on a computer is complicated by the fact that computer programs are inherently deterministic: while the output of computer program may look random, it is obtained by executing the steps of some algorithm and thus is totally predictable. The problem of generating random numbers is then split into two distinct subproblems:

1. generating any randomness at all, concentrating on the simple case of generating independent random numbers, uniformly distributed on $[0, 1]$ (Chapter 1);

2. generating random numbers from different distributions, using independent, uniformly distributed random numbers as a basis (Chapter 2).

Afterwards, we introduce the notion of random field, as a general tool to parametrize random *inputs* of models involving differential problems that will be addressed in the subsequent Parts. In particular, we show how the Karhunen-Loeve expansion, based on the eigendecomposition of a suitable compact operator, is of key importance to approximate random fields by means of finite-dimensional objects, thus allowing us to parametrize random fields, and making the ability to sample from statistical distributions even more important (Chapter 3).

Being able to generate random numbers, and sample from statistical distributions, is a building block of Monte Carlo algorithms that are addressed in Chapter 4. In particular, we introduce the Monte Carlo method, showing its properties, its connection with known quadrature formulas, after recalling some asymptotic results. In order to improve its convergence rate, we introduce in Chapter 5 a series of variance reduction techniques, such as antithetic variables, importance sampling, control variates, stratified sampling, latin hypercube sampling. Finally, we address in Chapter 6 some ideas related with quasi Monte Carlo methods, which rely on suitable alternatives to (pseudo) random sampling such as low-discrepancy points sets or sequences, and are capable of reaching a better convergence rate than Monte Carlo methods.

Chapter 1

Uniform (Pseudo) Random Number Generation

Why UNIFORM? Because as soon as we are able to sample from an uniform distribution we are also able to sample from other distributions (transforming variables,..) (distributions for which we know at least the cdf /pdf)

At the heart of any Monte Carlo method is a *random number generator*, i.e. a procedure that produces an infinite stream $U_1, U_2, U_3, \ldots$ μ of random variables that are independent and identically distributed (iid) according to some probability distribution μ. In particular, if μ is the uniform distribution on $(0,1)$, $\mu = \mathcal{U}(0,1)$, the generator is called a *uniform random number generator*.

Most computer languages already contain a built-in uniform random number generator. The user is typically requested only to input an initial number, called the *seed*, and upon invocation the random number generator produces a sequence of independent uniform random variables on the interval $(0,1)$. In `Matlab`, for example, this is provided by the `rand` function.

The concept of an infinite iid sequence of random variables is a mathematical abstraction that may be impossible to implement on a computer. The best one can hope to achieve in practice is to produce a sequence of *random* numbers with statistical properties that are indistinguishable from those of a true sequence of iid random variables. Although generators based on physical devices that exploit universal background radiation or quantum mechanics exist, the vast majority of current random number generators (RNG) are based on *algorithms* that can be implemented on a computer. As such, these algorithms produce a *purely deterministic* stream of numbers $U_1, U_2, U_3, \ldots$ which, however, resemble a stream of iid random variables in the sense that the stream is indistinguishable from a random one according to a number of statistical tests. Algorithmic generators are called *pseudorandom number generators*.

we'll use statistical tests that will verify how much random the generated numbers are

General structure of a pseudorandom number generator
A pseudo RNG is an algorithm which outputs a sequence of numbers that can be used as a replacement for an independent and identically distributed (i.i.d.) sequence of *true* random numbers. A pseudo RNG can be represented by a tuple $(S, f, \mu, \mathcal{U}, g)$ where:

- S is a finite set of states;
- $f : S \to S$ is a function defined on S;
- μ is a probability distribution on S;
- $\mathcal{U}$ is the output space; for a uniform generator, $\mathcal{U} = (0,1)$;
- $g : S \to \mathcal{U}$.

Example: a deterministic system can imitate a random phenomenum, but:
Consider for instance an iterative map:

$X_{n+1} = D(X_n)$, $D_\alpha(x) = \alpha x(1-x)$ logistic function : $\alpha \in [3.57, 4] \Rightarrow$ chaotic pattern

In particular, if $\alpha = 4$ we get a sequence $\{X_n\} \subset [0,1]$ that (theoretically) has the same behavior of a random variable distributed according to a Beta $(1/2, 1/2)$ (also called "arcsin" distribution), i.e. $f(x) = \left(\sqrt{\pi^2 x(1-x)}\right)^{-1}$ $x \in (0,1)$. The point is: if we plot (X_n, X_{n+100}) we have a behavior of an uniform distribution more than if we plot $(X_n, X_{n+1}) \Rightarrow$ there is dependence between the sample and the following sample. Should we discard 99 X_n? Too expensive! Moreover, because of the truncation effect (due to computer) we obtain periodic patterns, something that we don't want to meet when sampling randomly

A pseudo RNG has the following algorithmic structure:

Algorithm 1 Pseudo Random Number Generator

1: Draw the seed X_0 from the distribution μ on S
2: **for** $k = 1, 2, \ldots$ **do**
3: $X_k = f(X_{k-1})$ (recursion on state variable $X_k \in S$) $f : S \to S$
4: $U_k = g(X_k)$ (output $U_k \in \mathcal{U}$) $g : S \to \mathcal{U}$
5: **end for**

In particular:

- The initial state X_0 is called the *seed*; a RNG starting from a given seed will *always* produce the same sequence $U_1, U_2, \ldots$ – this is actually a convenient feature when testing or debugging a code.

- Since the state space S is finite, the generator eventually repeats itself, i.e., it revisits an already visited state. All pseudo RNG are periodic! We all *period* the smallest number of steps taken before visiting a previously visited state.

Some properties of a good uniform pseudo RNG are the following ones:

- *Large period*: if we need to run a Monte Carlo (MC) analysis using M (pseudo) random variables, the period l of the generator should be $l \gg M$, otherwise the property of independent samples is clearly broken. The period of a random number generator should then be extremely large – on the order of 10^{50} – in order to avoid problems with duplication and dependence. Most early algorithmic random number generators were fundamentally inadequate in this respect.

- *Pass a battery of statistical tests* for uniformity and independence. Since the generator should produce a stream of uniform random numbers that is indistinguishable from a genuine uniform iid sequence, practically speaking this means that the generator should pass a battery of simple statistical tests designed to detect deviations from uniformity and independence.

in the example of the previous page we said that we should discard 99 samples every sample to obtain a random sequence → very inefficient

- Fast and efficient: the generator should produce random numbers in a fast and efficient manner, and require little storage in computer memory. Many MC techniques require the generation of billions of random variables. In certain fields (e.g. finance) the generation time is a big issue.

- *Reproducible*: in certain cases it is important to be able to reproduce a stream $U_1, U_2, \ldots$ without the need of storing it (debugging purposes, advanced MC variance reduction techniques, ...).

- *Possibility of generating multiple streams.* In many applications it is necessary to run multiple independent random streams in parallel.

- *Avoid to produce the numbers 0 and 1.* This is to avoid division by 0 or other numerical complications.

1.1 Some common pseudo RNGs

The most commonly used generators are based on *linear* recurrences.

Linear Congruential Generator

A first example is provided by the *Linear Congruential Generator* (LCG):

Linear Congruential Generator (LCG)

- state space $S = \{0, 1, \dots, m-1\}$ (m is called *modulus*)
- recursion $X_k = (aX_{k-1} + b) \mod m$, $a, b \in \mathbb{N}$ $= f(X_{k-1})$
- output $U_k = X_k/m$. $= g(X_k)$

since we want values $\in [0,1]$ we divide $\forall X_k$ by m (where the maximum value of X_k is $m-1$)

Here the modulus $m > 0$, the *multiplier* $a \in (0, m)$ and the increment $b \in (0, m)$ are integer constants that specify the generator. If $b = 0$, the generator is often called a multiplicative congruential generator (MCG).

In the algorithm, *mod* denotes the modulus for integer division, that is the value $n \mod m$ is the remainder of the division of n by m, in the range $0, 1, \dots, m-1$. Thus the sequence generated by the recursion step consists of integers X_k from the range $\{0, 1, 2, \dots, m-1\}$.

While the output of the LCG looks random, from the way it is generated it is clear that the output has several properties which make it different from truly random sequences. For example, since each new value of X_k is computed from X_{k-1}, once the generated series reaches a value X_k which has been generated before, the output starts to repeat.

Since X_k can take only m different values, the output of a LCG starts repeating itself after at most m steps; the generated sequence is eventually periodic. Sometimes the periodicity of a sequence of pseudo random numbers can cause problems, but on the other hand, if the period length is longer than the amount of random numbers we use, periodicity cannot affect our result. For this reason, one needs to carefully choose the parameters m, a and b to achieve a long enough period. In particular m needs to be chosen large, being an upper bound for the period length.

LCGs were popular for many years but are now somewhat outdated (e.g. Matlab versions up to 5 were using them). LCG can generate any number in

$$\left\{0, \frac{1}{m}, \dots, \frac{m-1}{m}\right\}$$

where m^{-1} should be taken of the order of the floating point machine precision (ε-machine).

- **Example 1.1.1.** *A popular choice is the so-called Lewis-Goodman-Miller LCG, where* $a = 7^5 = 16\,807$, $b = 0$, $m = 2^{31} - 1 \approx 2 \cdot 10^9$, *with a resulting maximal period too small for today's applications. Indeed, although this LCG passes many of the standard statistical tests and has been successfully used in many applications – and for this reason it is sometimes viewed as the* minimal *standard LCG, against which other generators should be judged – its period (and statistical properties) no longer meet the requirements of modern Monte Carlo applications.*

Matrix Congruential Generator

Another option is the so-called *Matrix Congruential Generator* (MCG) of order q, which can be seen as the q- dimensional generalization of the multiplicative congruential generator.

Matrix Congruential Generator (MCG) of order q

- state space $S = \{0, 1, \ldots, m-1\}^q$
- state variable $\mathbf{X} = (X^1, \ldots, X^q)^\top \in S$
- recursion $\mathbf{X}_k = A\mathbf{X}_{k-1} \mod m, \qquad A \in \mathbb{N}^{q \times q}$, invertible
- output $\mathbf{U}_k = \mathbf{X}_k / m \in (0,1)^q$.

once we obtain the vector $A\mathbf{X}_{k-1}$ then we proceede component wise with **mod m**

For the sake of efficiency, A must be sparse (i.e., it must have a lot of entries equal to 0). A special case of a matrix congruential generator is given by the choice

$$A = \begin{pmatrix} 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 \\ a_q & a_{q-1} & \ldots & a_1 \end{pmatrix}$$

also here the division is componentwise. we obtain a vector of iid $\mathcal{U}(0,1)$ random numbers; we can sum the numbers increasing the randomicity

which leads to the recursion on $X_k \in \{0, 1, \ldots, m-1\}$:

$$X_k = (a_1 X_{k-1} + a_2 X_{k-2} + \ldots + a_q X_{k-q}) \mod m, \qquad U_k = \frac{X_k}{m}.$$

This is called a *multiple recursive generator* (MRG) of order q, and its maximum period can be up to $m^q - 1$. MRGs with very large periods can be implemented efficiently by combining several smaller-period MRGs, see Sect. 1.2.

Modulo 2 linear generators

Good random generators must have very large state spaces. For an LCG this means that the modulus m must be a large integer. However, for multiple recursive generators and matrix generators, it is not necessary to take a large modulus, as the state space can be as large as m^q . Because binary operations are in general faster than floating point operations (which are in turn faster than integer operations), it makes sense to consider random number generators that are based on linear recurrences modulo 2.

Modulo 2 linear generators are Matrix congruential generators with modulus $m = 2$; to have long periods, the order q has to be large (the maximal period is $2^q - 1$). An example within this class is the *Linear Feedback Shift Register* (LFSR) generator, also called Tausworthe. It is in the form of a MRG with $m = 2$,

$$X_k = (a_1 X_{k-1} + \ldots + a_q X_{k-q}) \mod 2$$

and each word $(X_0, \ldots, X_{w-1})$, $(X_w \ldots, X_{2w-1}), \ldots$ is interpreted as a binary representation of a number and then we recover a number between 0 and 1 as:

$$U_k = \sum_{k=1}^{w} X_{kw+l-1} 2^{-l}.$$

For fast generation, most of the a_j are zero. In many cases there is only one non zero multiplier a_r apart from a_q, in which case $X_k = X_{k-r} \oplus X_{k-q}$, where $\oplus$ is the addition modulo z.

Generalizations of the LSFR include the Mersenne Twister (1997), that is now the default generator in `Matlab` and `R` and avoids many of the problems with earlier generators. **Hands-On Problem 1.1** focuses on the properties of the `rand` command in `Matlab`. The Mersenne Twister has a period of $2^{19937} - 1$, is very fast, and passes all statistical tests. Moreover, it is proven to be equidistributed in (up to) 623 dimensions (for 32-bit values), and at the time of its introduction was running faster than other statistically reasonable generators. Curious about the code? Visit `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html`

1.2 Combined generators

The idea of *combined generators* is to combine the output of several generators which, individually, may be of poor quality, to make a superior quality generator.

An example is provided by the *Wichmann-Hill* generator, proposed in 1982 by Brian Wichmann and David Hill. It consists of three linear congruential generators with different prime moduli, each of which is used to produce a uniformly distributed number between 0 and 1. These are summed, modulo 1, to produce the result:

$$\left.\begin{array}{lll} X_k = (171X_{k-1}) & \mod m_1 & (m_1 = 30\,269) \\ Y_k = (172Y_{k-1}) & \mod m_2 & (m_2 = 30\,307) \\ Z_k = (170Z_{k-1}) & \mod m_3 & (m_3 = 30\,323) \end{array}\right\} \quad U_k = decimal\left(\frac{X_k}{m_1} + \frac{Y_k}{m_2} + \frac{Z_k}{m_3}\right)$$

In the last operation, we take the decimal part of the number[1] Summing three generators produces a pseudorandom sequence with period $l \approx 6.95 \times 10^{12}$. Specifically, the moduli are 30269, 30307 and 30323, producing periods of 30268, 30306 and 30322. The overall period is the least common multiple of these: $30268 \times 30306 \times 30322/4 = 6953607871644$. Although it is not very large for today's applications, the Wichmann-Hil generator performs quite well in simple statistical tests.

Remark 1.2.1. *One class of combined generators that has been extensively studied is that of the combined multiple-recursive generators (MRGs), where a small number of MRGs are combined.*

Example 1.2.1. *Another example is the MRG32k3a generator, a 32-bit implementation of L'Ecuyer's combined multiple recursive generator (1999), which is given by the combination of 2 MRGs:*

$$\left.\begin{array}{l} X_k = (a_2X_{k-2} + a_3X_{k-3}) \mod m_1 \\ Y_k = (b_1Y_{k-1} + b_3Y_{k-3}) \mod m_2 \end{array}\right\} \quad U_k = \begin{cases} \frac{X_k - Y_k + m_1}{m_1+1} & X_k \le Y_k \\ \frac{X_k - Y_k}{m_1+1} & X_k > Y_k \end{cases}$$

with suitable values of $a_2, a_3, b_1, b_3, m_1, m_2$. It has a period $l \approx 3 \times 10^{57}$ and passes all statistical tests; it has been implemented in many packages including `Matlab`, `Mathematica`, *Intel's MKL Library, ... A* `Matlab` *implementation is reported below.*

```
&MRG32k3a.m
m1=2^32-209; m2=2^32-22853;
ax2p=1403580; ax3n=810728;
ay1p=527612;    ay3n=1370589;

X= [12345 12345 12345]; % Initial X
Y= [12345 12345 12345]; % Initial Y

N=100; % Compute the sequence for N steps
U=zeros(1,N);
for t=1:N
    Xt=mod(ax2p*X(2)-ax3n*X(3),m1);
    Yt=mod(ay1p*Y(1)-ay3n*Y(3),m2);
    if Xt <= Yt
        U(t)=(Xt - Yt + m1)/(m1+1);
    else
        U(t)=(Xt - Yt)/(m1+1);
    end
    X(2:3)=X(1:2); X(1)=Xt; Y(2:3)=Y(1 2); Y(1)=Yt;
end
```

[1] For example, if the three WH equations give 0.68, 0.92, and 0.35, then the final result random number is decimal(0.78 + 0.92 + 0.45) = decimal(2.95) = 0.95. The decimal part of `x` can be computed in `Matlab` by `x-fix(x)`; `fix(x)` rounds the elements of x to the nearest integer towards zero.

Is the produced sequence good? Does it resemble to an iid sample from $\mathcal{U}([0,1])$?

1.3 Empirical Tests for RNG

The quality of random number generators can be assessed either by means of *theoretical tests*, investigating the theoretical properties of the RNG without requiring the actual output of the generator but only its algorithmic structure and parameters, or through *empirical tests*, involving the application of a battery of statistical tests to the output of the generator, with the objective to detect deviations from uniformity and independence. Here we focus on this latter class.

Several statistical tests have been proposed to assess the quality of a RNG. Today's most comprehensive test suite is `TestU01` developed by L'Ecuyer and Simard (2007). Before mentioning some examples, let us review some nonparametric Goodness-of-fit Tests.

Review of nonparametric Goodness-of-fit Tests

Let X be a random variable with values in $I \subset \mathbb{R}$, probability density function (PDF)

$$f : I \to \mathbb{R}_+, \qquad \int f(x)dx = 1$$

and (absolutely) continuous, cumulative distribution function (CDF)

$$F(x) = \int_{-\infty}^{x} f(t)dt.$$

We will often omit the subscript X in $f_X(\cdot)$ and $F_X(\cdot)$ and write $f(\cdot)$ and $F(\cdot)$, respectively.

Let $\mathbf{X} = (X_1, \dots, X_n)$ be a random sample. We want to test the hypothesis H_0 that $\mathbf{X}$ has been drawn independently from the distribution $F(\cdot)$. Let moreover $\hat{F}_n(x)$ be the empirical distribution function,

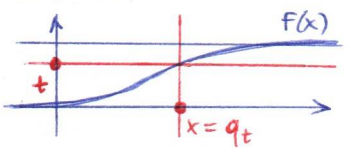
$$\hat{F}_n(x) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}_{\{X_i \le x\}} = \frac{\#\{X_i \le x\}}{n} = \frac{\text{number of elements in the sample} \le x}{n}$$

where[2] $\mathbf{1}_A$ is the indicator of event A. This cumulative distribution function is a step function that jumps up by $1/n$ at each of the n data points. Its value at any specified value of the measured variable is the fraction of observations of the measured variable that are less than or equal to the specified value.

Q-Q plot

A first simple graphical test to see if the sample $\mathbf{X}$ has been drawn from the distribution $F(\cdot)$ is to plot the quantiles of $\hat{F}_n(\cdot)$ versus the corresponding quantiles of $F(\cdot)$, where the quantiles of $F(\cdot)$ are

$$q_t = \arg\min_x \{F(x) \ge t\}$$



while the quantiles of $\hat{F}_n(\cdot)$ are, for any $j = 1, 2, \dots, n$,

$$\hat{q}_{j/n} = \arg\min_x \{\hat{F}_n(x) \ge j/n\} = X^{(j)},$$

that is, the j-th ordered value in the sample $\mathbf{X}$; a better estimate is actually $\hat{q}_{\frac{j}{n+1}} = X^{(j)}$. Hence, we can plot $\hat{q}_{\frac{j}{n+1}}$ versus $q_{\frac{j}{n+1}}$: if the sample $\mathbf{X}$ is drawn independently from $F(\cdot)$, the points should be well aligned on the diagonal.

Kolmogorov-Smirnov test

[2] For a fixed x, the indicator $\mathbf{1}_{\{X_i \le x\}}$ is a Bernoulli random variable with parameter $p = F(x)$; hence $n\hat{F}_n(x)$ is a binomial random variable with mean $nF(x)$ and variance $nF(x)(1 - F(x))$. This implies that $\hat{F}_n(x)$ is an unbiased estimator for $F(x)$.

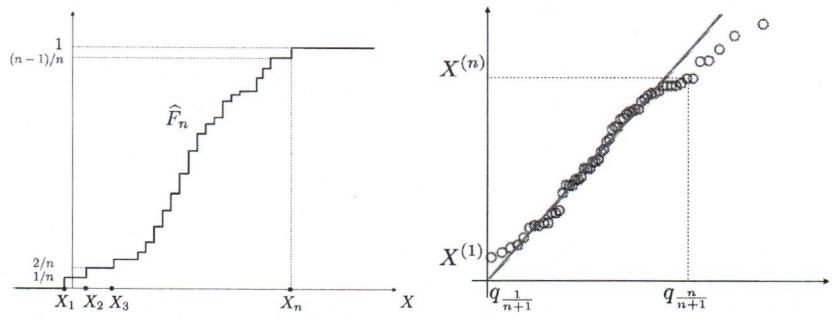


Figure 1.1: Left: cumulative distribution function. Right: Q-Q plot.

By the strong law of large numbers, the estimator $\widehat{F}_n(x)$ converges to $F(x)$ as $n \to \infty$ almost surely, for every value of x:

$$\widehat{F}_n(x) \xrightarrow{\text{a.s.}} F(x);$$

thus the estimator $\widehat{F}_n(x)$ is consistent. This expression asserts the pointwise convergence of the empirical distribution function to the true CDF. There is a stronger result, called the Glivenko-Cantelli theorem, which states that the convergence in fact happens uniformly over x:

$$\|\widehat{F}_n - F\|_\infty \equiv \sup_{x\in\mathbb{R}} |\widehat{F}_n(x) - F(x)| \xrightarrow{\text{a.s.}} 0.$$

The sup-norm in this expression is called the Kolmogorov-Smirnov statistic for testing the goodness-of-fit between the empirical distribution $\widehat{F}_n(x)$ and the assumed true CDF $F(x)$.

The Kolmogorov-Smirnov test is a more quantitative test that compares the empirical distribution $\hat{F}_n(\cdot)$ with the true one $F(\cdot)$.

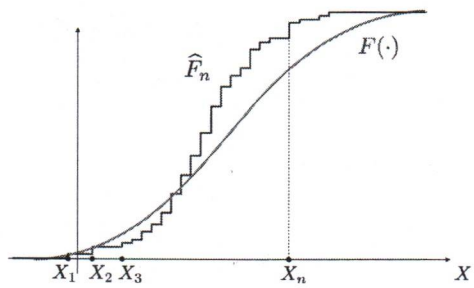


Figure 1.2: Empirical distribution $\hat{F}_n(\cdot)$ and *true* distribution $F(\cdot)$.

Let us define by

$$D_n = \sup_{x\in\mathbb{R}} |\widehat{F}_n(x) - F(x)|;$$

D_n = Kolmogorov-Smirnov statistic

clearly, D_n is a random variable as it depends on the random sample $\mathbf{X}$. For a continuous distribution $F(\cdot)$; and under the null hypothesis H_0 that the sample comes from the hypothesized distribution F, it is known (this result may also be known as the *Kolmogorov theorem*) that[3]

$$\sqrt{n}D_n \rightharpoonup K \qquad \text{independently of } F,$$

Kolmogorov random variable

[3] $X_n \rightharpoonup Y$, convergence in distribution (or weakly convergence, or convergence in law), means that $\lim_{n\to\infty} F_n(x) = F(x)$, $\forall x \in \mathbb{R}$ at which F is continuous. Here F_n and F are the CDFs of random variables X_n and X, respectively.

where K is a Kolmogorov random variable [4]

$$F_K(x) = P(K \leq x) = \left(1 + 2\sum_{j=1}^{\infty}(-1)^j e^{-2j^2x^2}\right)\mathbf{1}_{x>0}$$

and corresponds to the distribution of

$$\sup_{t\in[0,1]} |B(t)|,$$

where $B(t)$ is the Brownian bridge in $[0,1]$. This result shows that, under H_0, $F_n(\cdot) \to F(\cdot)$ at a rate $O(1/\sqrt{n})$ in a probabilistic sense.

Based on this result, the goodness-of-fit test (or the Kolmogorov-Smirnov test) can be constructed by using the critical values of the Kolmogorov distribution, and we can *reject* H_0 at level α if

$$\sqrt{n}D_n > K_\alpha,$$

with K_α : $P(K \leq K_\alpha) = 1 - \alpha$; the quantiles K_α are tabulated.

Back to **RNG**: equidistribution test: we test if $\{U_i\}_i$ (the generated "random" numbers) has a uniform distribution by using K-S test

χ^2 Test

Let X be a continuous random variable taking values in $I \subset \mathbb{R}$, with density $f(\cdot)$ and cumulative distribution function $F(\cdot)$. Let $\mathbf{X} = (X_1, \ldots, X_n)$ be a random sample. To test the hypothesis H_0 that $\mathbf{X}$ has been drawn independently from the distribution $F(\cdot)$, we split I in $k+1$ non-overlapping subintervals (classes) I_j, $j = 1, \ldots, k+1$ such that

the idea is to compare the distributions on the histograms instead of the cumulative distr. functions: we check whether the histogram of the sample is similar to the one of the true (theoretical) distribution

$$\bigcup_{j=1}^{k+1} = I, \qquad \mathring{I}_j \cap \mathring{I}_k = \emptyset.$$

Then, for each j, we define

$$N_j = \sum_{i=1}^{n} \mathbf{1}_{X_i \in I_j} = \#\{X_i \text{ fall in } I_j\},$$

$$p_j = P(X \in I_j) = \int_{I_j} f(x)dx;$$

under H_0, $\mathbb{E}[N_j] = np_j$. Indeed, np_j is the expected (theoretical) count of type j, asserted by the null hypothesis that the fraction of type j in the population is np_j.

We can then define the statistics

$$Q_k = \sum_{j=1}^{k+1} \frac{(N_j - np_j)^2}{np_j}$$

[4] A *Brownian bridge* is a continuous-time stochastic process $B(t)$ whose probability distribution is the conditional probability distribution of a Wiener process $W(t)$ (a mathematical model of Brownian motion) subject to the condition (when standardized) that $W(T) = 0$, so that the process is pinned at the origin at both $t = 0$ and $t = T$. More precisely: $B_t := (W_t \mid W_T = 0)$, $t \in [0, T]$. The expected value of the bridge is zero, with variance $t(T-t)/T$, implying that the most uncertainty is in the middle of the bridge, with zero uncertainty at the nodes. The covariance of $B(s)$ and $B(t)$ is $s(T-t)/T$ if $s < t$. The increments in a Brownian bridge are not independent. The Wiener process W_t is characterized by the following properties:

- $W_0 = 0$;
- W has independent increments: for every $t > 0$, the future increments $W_{t+u} - W_t$, $u \geq 0$, are independent of the past values W_s, $s \leq t$;
- W has Gaussian increments: $W_{t+u} - W_t \sim \mathcal{N}(0, u)$;
- W has continuous paths, that is, W_t is continuous in t.

as a normalized sum of squared deviations between observed and theoretical frequencies; Q_k has an asymptotic $\chi^2(k)$ distribution with k degrees of freedom (equal to the number of classes minus 1). We can then reject the null hypothesis H_0 at level α if

$$Q_k > q_{1-\alpha},$$

where $q_{1-\alpha}$ is the $1-\alpha$ quantile of the $\chi^2(k)$ distribution. Notice that (I_j, N_j) define a histogram of the sample and Q_k estimates the deviation from the *true* histogram (I_j, np_j).

Examples of empirical tests for RNGs
Below we report some examples of empirical tests for RNGs:

1. *Equidistribution test.* We test whether $\{U_i\}_{i=0}^{n-1}$ has a uniform distribution $\mathcal{U}(0,1)$. We can apply the Kolmogorov-Smirnov test or the $\boldsymbol{\chi}^2(m-1)$ test on the partition

$$[0, 1/m), \quad [1/m, 2/m), \quad \ldots, \quad [(m-1)/m, 1].$$

2. *Serial test.* We test whether successive values are uniformly distributed. Namely, we group the $\{U_i\}_{i=0}^{n-1}$ in groups of length d: $\mathbf{U}_1 = (U_0, \ldots, U_{d-1})$, $\mathbf{U}_2 = (U_d, \ldots, U_{2d-1}), \ldots$ and test whether $\{\mathbf{U}_j\}$ has a multivariate uniform distribution $U((0,1)^d)$ by a $\boldsymbol{\chi}^2$ test on the partition

$$I_{jk} = \left[\frac{j-1}{m}, \frac{j}{m}\right] \times \left[\frac{k-1}{m}, \frac{k}{m}\right], \qquad j, k = 1, \ldots, m.$$

   Of course, n should be sufficiently large compared to m^2 so that each class has enough samples and one can apply the asymptotic result.

3. *Gap test.* Let $T_1, T_2, \ldots$ the times when the process $\{U_i\}_{i=0}^{n-1}$ visits a given interval $(\alpha, \beta \subset (0,1)$, namely $U_{T_j} \in (\alpha, \beta)$, $U_K \notin (\alpha, \beta)$, $K \notin \{T_1, T_2, \ldots\}$. Let $Z_i = T_i - T_{i-1}$ be the gap length between two consecutive visits (here $T_0 = 0$). Under H_0, Z_i are iid with a geometric distribution with parameter $p = \beta - \alpha$, that is,

$$P(Z = j) = p(1-p)^j, \qquad j = 0, 1, 2, \ldots.$$

   The gap test assesses this hypothesis by counting the number of gaps that fall in certain classes. One can use a $\boldsymbol{\chi}^2(m)$ test to test whether the $\{Z_i\}$ have the right geometric distribution[5], using the classes $Z = 0$, $Z = 1$, $Z = r-1$, $Z \geq r$, with probabilities $p(1-p)^z$, $z = 0, \ldots, r-1$ for the first r classes and $(1-p)^r$ for the last class.

[5]The geometric distribution models the number of failures until the first success. If the probability of success on each trial is p, the number Z of failures until the first success is distributed according to $P(Z = k) = (1-p)^k p$, for $k = 1, 2, \ldots$.