Contract no: 20851A2/1/143

Random Numbers

Pierre L'Ecuyer

Université de Montréal, Montréal, Québec, Canada

Random numbers generators (RNGs) are available from many computer software libraries. Their purpose is to produce sequences of numbers that *appear* as if they were generated randomly from a specified probability distribution. These *pseudorandom numbers*, often called *random numbers* for short (with a slight abuse of language), are crucial ingredients for a whole range of computer usages, such as statistical experiments, simulation of stochastic systems, numerical analysis with Monte Carlo methods, probabilistic algorithms, computer games, cryptography, and gambling machines, to name a few. These RNGs are small computer programs implementing carefully crafted algorithms, whose design is (or should be) based on solid mathematical analysis. Usually, there is a basic *uniform* RNG whose aim is to produce numbers that imitate independent random variables from the uniform distribution over the interval $[0, 1]$ (i.i.d. U[0,1]), and the random variables from other distributions (normal, chi-square, exponential, Poisson, etc.) are simulated by applying appropriate transformations to the uniform random numbers.

1 Random number generators

The words *random number* actually refer to a poorly defined concept. Formally speaking, one can argue that no number is more random than any other. If a fair die is thrown 8 times, for example, the sequences of outcomes 22222222 and 26142363 have exactly the same chance of occuring, so on what basis could the second one be declared more random than the first?

A more precisely defined concept is that of independent random variables. This is a pure mathematical abstraction which is nevertheless convenient for modeling real-life situations. The outcomes of successive throws of a fair die, for instance, can be modeled as a sequence of independent random variables uniformly distributed over the set of integers from 1 to 6, i.e., where each of these integers has probability 1/6 for each outcome, independently of the other outcomes. This is an abstract concept because no absolutely perfect die exists in the real world.

A *Random number generator* (RNG) is a computer program whose aim is to *imitate* or *simulate* the typical behavior of a sequence of independent random variables. Such a program is usually purely deterministic: if started at different times or on different computers with the same initial state of its internal data, i.e., with the same *seed*, the sequence of numbers produced by the RNG will be exactly the same.

When constructing an RNG to simulate a die, for example, the aim is that the statistical properties of the sequence of outcomes be representative of what should be expected from an idealized die. In particular, each number should appear with frequency near 1/6 in the long run, each pair of successive numbers should appear with frequency near 1/36, each triplet should appear with frequency near 1/216, and so on. This would guarantee not only that the virtual die is unbiased, but also that the successive outcomes are uncorrelated.

However, practical RNGs use a finite amount of memory. The number of states that they can take is finite, and so is the total number of possible streams of successive output values that they can produce, from all possible initial states.

Mathematically, an RNG can be defined (see L'Ecuyer 1994) as a structure (S, μ, f, U, g), where S is a finite set of *states*, μ is a probability distribution on S used to select the *initial state* (or *seed*) s_0, the mapping $f : S \to S$ is the *transition function*, U is a finite set of *output* symbols, and $g : S \to U$ is the *output function*.

The state evolves according to the recurrence $s_n = f(s_{n-1})$, for $n \geq 1$. The *output* at step n is $u_n = g(s_n) \in U$. These u_n are the so-called *random numbers* produced by the RNG. Because S is finite, the generator will eventually return to a state already visited (i.e., $s_{i+j} = s_i$ for some $i \geq 0$ and $j > 0$). Then, $s_{n+j} = s_n$ and $u_{n+j} = u_n$ for all $n \geq i$. The smallest $j > 0$ for which this happens is called the *period* length ρ. It cannot exceed the cardinality of S. In particular, if b bits are used to represent the state, then $\rho \leq 2^b$. Good RNGs are designed so that their period length is close to that upper bound.

Choosing s_0 from a uniform initial distribution μ can be crudely approximated by using external randomness such as picking balls from a box or throwing a real die. The RNG stretches a short random seed into a long stream of random-looking numbers.

2 Randomness from physical devices

Why use a deterministic computer algorithm instead of a truly random mechanism for generating random numbers? Simply calling a system's function from a program is certainly more convenient than throwing dice or picking balls from a box and entering the corresponding numbers on a computer's keyboard, especially when thousands (often millions or even billions) of random numbers are needed for a computer experiment.

Attempts have been made at constructing RNGs from physical devices such as noise diodes, gamma-ray counters, and so on, but these remain largely impractical and unreliable, because they are cumbersome, and because it is generally not true that the successive numbers that they produce are independent and uniformly distributed. If one insists on getting away from a purely deterministic algorithm, a sensible compromise is to combine the output of a well-designed RNG with some physical noise (Marsaglia 1985; L'Ecuyer 1998).

3 Quality criteria

Assume, for the following, that the generator's purpose is to *simulate* i.i.d. $U[0,1]$ random variables. The goal is that one cannot easily distinguish between the output sequence of the RNG and a sequence of i.i.d. $U[0,1]$ random variables. That is, the RNG should behave according to the null hypothesis $\mathbb{H}_0$: "The u_n are i.i.d. $U[0,1]$", which means that for each t, the vector $(u_0,\ldots,u_{t-1})$ is uniformly distributed over the t-dimensional unit cube $[0,1]^t$. Clearly, $\mathbb{H}_0$ cannot be exactly true, because these vectors always take their values only from the finite set

$$\Psi_t = \{(u_0,\ldots,u_{t-1}) : s_0 \in S\}.$$

If s_0 is random, Ψ_t can be viewed as the *sample space* from which the output vectors are taken. The idea then is to require that Ψ_t be very evenly distributed over the unit cube, so that $\mathbb{H}_0$ be *approximately true for practical purposes*, at least for moderate values of t. A huge state set S and a very large period length ρ, much much larger than any value of t of interest, are *necessary* requirements, but they are not sufficient; good uniformity of the Ψ_t's is also needed. When several t-dimensional vectors are produced by an RNG by taking non-overlapping blocks of t output values, this can be viewed in a way as picking points at random from Ψ_t, without replacement.

It remains to agree on a computable *figure of merit* that measures the evenness of the distribution, and to construct specific RNGs having a good figure of merit in all dimensions t up to some preset number t_1 (because computing the figure of merit for all t up to infinity is infeasible in general). Several measures of *discrepancy* between the empirical distribution of a point set Ψ_t and the uniform distribution over $[0,1]^t$ have been proposed and studied over the last few decades (Niederreiter 1992; Hellekalek and Larcher 1998). These measures can be used to define figures of merit for RNGs. A very important criterion in choosing such a measure is the ability to compute it efficiently, and this depends on the mathematical structure of Ψ_t. For this reason, it is customary to use different figures of merit (i.e., different measures of discrepancy) for analyzing different classes of RNGs. Some would argue that Ψ_t should look like a typical set of random points over the unit cube instead of being too evenly distributed, i.e., that it should have a chaotic structure, not a regular one. However, chaotic structures are hard to analyze mathematically. It is probably safer to select RNG classes for which the structure of Ψ_t can be analyzed and understood, even if this implies more regularity, rather than selecting an RNG with a chaotic but poorly understood structure.

For a concrete illustration, consider the two linear recurrences

$$
\begin{aligned}
x_{1,n} &= (a_{1,1}x_{1,n-1} + \cdots + a_{1,k}x_{1,n-k}) \bmod m_1,\\
x_{2,n} &= (a_{2,1}x_{2,n-1} + \cdots + a_{2,k}x_{2,n-k}) \bmod m_2,
\end{aligned}
$$

where k, the m_j's, and the $a_{i,j}$'s are fixed integers, "mod m_j" means the remainder of the integer division by m_j, and the output function

$$u_n = (x_{1,n}/m_1 - x_{2,n}/m_2) \bmod 1.$$

This RNG is a *combined linear multiple recursive generator* (L'Ecuyer 1999a). Its state at step n is the $2k$-dimensional vector $(x_{1,n}, \ldots, x_{1,n-k+1}, x_{2,n}, \ldots, x_{2,n-k+1})$, whose first k components are in $\{0, 1, \ldots, m_1 - 1\}$ and last k components are in $\{0, 1, \ldots, m_2 - 1\}$. There are thus $(m_1 m_2)^k$ different states and it can be proved that Ψ_k is the set of all k-dimensional vectors with coordinates in $\{0, 1/m, \ldots, (m-1)/m\}$, where $m = m_1 m_2$. This implies, if m is large, that for each $t \leq k$, the t-dimensional cube $[0, 1]^t$ is evenly covered by Ψ_t. For $t > k$, it turns out that in this case, Ψ_t has a regular lattice structure (Knuth 1998; L'Ecuyer 1998), so that all its points are contained in a series of equidistant parallel hyperplanes, say at a distance d_t apart. To obtain a good coverage of the unit cube, d_t must be as small as possible, and this value can be used as a selection criterion.

L'Ecuyer (1999a) recommends a concrete implementation of this RNG with the parameter values: $k = 3$, $m_1 = 2^{32} - 209$, $(a_{11}, a_{12}, a_{13}) = (0, 1403580, -810728)$, $m_2 = 2^{32} - 22853$, and $(a_{11}, a_{12}, a_{13}) = (527612, 0, -1370589)$. This RNG has two main cycles of length $\rho \approx 2^{191}$ each (the total number of states is approximately 2^{192}). Its lattice structure has been analyzed in up to $t = 48$

dimensions and was found to be excellent, in the sense that d_t is not far from the smallest possible value that can be achieved for an arbitrary lattice with $(m_1 m_2)^3$ points per unit of volume. These parameters were found by an extensive computer search.

For a second example, consider the *linear congruential generator* (LCG) defined by the recurrence $x_n = ax_{n-1} \bmod m$ and output function $u_n = x_n/m$. This type of RNG is widely used and is still the default generator in several (perhaps the majority of) software systems. With the popular choice of $m = 2^{31} - 1$ and $a = 16807$ for the parameters, the period length is $2^{31} - 2$, which is now deemed much too small for serious applications (Law and Kelton 2000; L'Ecuyer 1998). This RNG also has a lattice structure similar to the previous one, except that the distances d_t are much larger, and this can easily show up in simulation results (Hellekalek and Larcher 1998; L'Ecuyer and Simard 2000). Small LCGs of this type should be discarded.

Another way of assessing uniformity is in terms of the leading bits of the successive output values, as follows. Suppose that S (and Ψ_t) has cardinality 2^e. There are $t\ell$ possibilities for the ℓ most significant bits of t successive output values $u_n, \ldots, u_{n+t-1}$. If each of these possibilities occurs exactly $2^{e-t\ell}$ times in Ψ_t, for all ℓ and t such that $t\ell \leq e$, the RNG is called *maximally equidistributed* (ME). Explicit implementations of ME or nearly ME generators can be constructed by using linear recurrences modulo 2 (L'Ecuyer 1999b; Tezuka 1995; Matsumoto and Nishimura 1998).

Besides having a long period and good uniformity of Ψ_t, RNGs must be *efficient* (run fast and use little memory), *repeatable* (the ability of repeating exactly the same sequence of numbers is a major advantage of RNGs over physical devices, e.g., for program verification and variance reduction in simulation; Law and Kelton 2000), and *portable* (work the same way in different software/hardware environments). The availability of efficient methods for jumping ahead in the sequence by a large number of steps is also an important asset of certain RNGs. It permits one to partition the sequence into long disjoint substreams and to create an arbitrary number of *virtual generators* (one per substream) from a single backbone RNG (Law and Kelton 2000).

Cryptologists use different quality criteria for RNGs (Knuth 1998; Lagarias 1993). Their main concern is *unpredictability* of the forthcoming numbers. Their analysis of RNGs is usually based on asymptotic analysis, in the framework of computational complexity theory. This analysis is for *families* of generators, not for particular instances. They study what happens when an RNG of size k (i.e., whose state is represented over k bits of memory) is selected randomly from the family, for $k \to \infty$. They seek RNGs for which the work needed to guess the next bit of output significantly better than by flipping a fair coin increases faster, asymptotically, than any polynomial function of k. This means that an intelligent guess quickly becomes impractical when k is increased. So far, generator families *conjectured* to have these properties have been constructed, but proving even the existence of families with these properties remain an open problem.

4 Statistical testing

RNGs should be constructed based on a sound mathematical analysis of their structural properties. Once they are constructed and implemented, they are usually submitted to *empirical statistical tests* that try detecting statistical deficiencies (see *Significance, tests of*; *Goodness of fit: overview*) by looking for empirical evidence against the hypothesis $\mathbb{H}_0$ defined previously. A test is defined by a test statistic T, function of a fixed set of u_n's, whose distribution under $\mathbb{H}_0$ is known. The number of different tests that can be defined is infinite and these different tests detect different problems with the RNGs. There is no universal test or battery of tests that can guarantee, when passed, that a given generator is fully reliable for all kinds of simulations. Passing a lot of tests may improve one's confidence in the RNG, but never proves that the RNG is foolproof. In fact, no RNG can pass all statistical tests. One could say that a *bad* RNG is one that fails *simple* tests, and a *good* RNG is one that fails only complicated tests that are very hard to find and run. In the end, there is no definive answer to the question "what are the good tests to apply?"

Ideally, T should mimic the random variable of practical interest in a way that a bad structural interference between the RNG and the problem will show up in the test. But this is rarely practical. This cannot be done, for example, for testing RNGs for general purpose software packages. For a sensitive application, if one cannot test the RNG specifically for the problem at hand, it is a good idea to try RNGs from totally different classes and compare the results.

Specific statistical tests for RNGs are described in Knuth (1998), Hellekalek and Larcher (1998), Marsaglia (1985), and other references given there. Experience with empirical testing tells us that RNGs with very long periods, good structure of their set Ψ_t, and based on recurrences that are not too simplistic, pass most reasonable tests, whereas RNGs with short periods or bad structures are usually easy to crack by statistical tests.

5 Non-uniform random variates

Random variates from distributions other than the uniform over $[0, 1]$ (see *Distributions, statistical*) are generated by applying further transformations to the output values u_n of the uniform RNG. For example, $y_n = a + (b - a)u_n$ produces a random variate uniformly distributed over the interval $[a, b]$. To generate a random variate y_n that takes the values -1 and 1 with probability 0.2 each, and 0 with probability 0.6, one can define $y_n = -1$ if $u_n \leq 0.2$, $y_n = 1$ if $u_n > 0.8$, and $y_n = 0$ otherwise.

Things are not always so easy, however. For certain distributions, compromises must be made between simplicity of the algorithm, quality of the approximation, robustness with respect to the distribution parameters, and efficiency (generation speed, memory requirements, and setup time). Generally speaking, one should favor simplicity over small speed gains.

Conceptually, the simplest method for generating a random variate y_n from distribution F is *inversion*: define $y_n = F^{-1}(u_n) \stackrel{\text{def}}{=} \min\{y \mid F(y) \geq u_n\}$. Then, if we view u_n as a uniform random variable, $P[y_n \leq y] = P[F^{-1}(u_n) \leq y] = P[u_n \leq F(y)] = F(y)$, so y_n has distribution F. Computing this y_n requires a formula or a good numerical approximation for F^{-1}. As an example, if y_n is to have the geometric distribution with parameter p, i.e., $F(y) = 1 - (1-p)^y$ for $y = 1, 2, \ldots$, the inversion method gives $y_n = F^{-1}(u_n) = 1 + \lfloor \ln(1-u_n)/\ln(1-p) \rfloor$. The normal, student, and chi-square are examples of distributions for which there are no close form expression for F^{-1}, but good numerical approximations are available (Bratley, Fox, and Schrage 1987).

In most simulation applications, inversion should be the method of choice, because it is a *monotone* transformation of u_n into y_n, which makes it compatible with major variance reductions techniques (Bratley, Fox, and Schrage 1987). But there are also situations where *speed* is the real issue and where monotonicity is no real concern. Then, it might be more appropriate to use fast non-inversion methods such as, for example, the *alias method* for discrete distributions or its continuous *acceptance-complement* version, the *acceptance/rejection* method and its several variants, *composition* and *convolution* methods, and several other specialized and fancy techniques often designed for specific distributions (Bratley, Fox, and Schrage 1987; Devroye 1986; Gentle 1998).

Recently, there has been an effort in developping so-called *automatic* or *black box* algorithms for generating variates from an arbitrary (known) density, based on acceptance/rejection with a transformed density (Hörmann and Leydold 2000). Another important class of general non-parametric methods is those that sample directly from a smoothened version of the empirical distribution of a given data set (Law and Kelton 2000; Hörmann and Leydold 2000). These methods shortcut the fitting of a specific type of distribution to the data.

6 A consumer's guide

No uniform RNG can be guaranteed against all possible defects, but the following ones have fairly good theoretical support, have been extensively tested, and are easy to use: the Mersenne twister of Matsumoto and Nishimura (1998), the combined MRGs of L'Ecuyer (1999a), the combined LCGs of L'Ecuyer and Andres (1997), and the combined Tausworthe generators of L'Ecuyer (1999b). Further discussion and up-to-date developments on RNGs can be found in Knuth (1998), L'Ecuyer (1998) and from the web pages:

`http://www.iro.umontreal.ca/~lecuyer`

`http://random.mat.sbg.ac.at`

`http://cgm.cs.mcgill.ca/~luc/`

`http://www.webnz.com/robert/rng_links.htm`

For non-uniform random variate generation, software and related information can be found at

`http://cgm.cs.mcgill.ca/~luc/`

`http://statistik.wu-wien.ac.at/src/unuran/`

Bibliography

Bratley, P., B. L. Fox, and L. E. Schrage. 1987. *A Guide to Simulation.* Second ed. New York: Springer-Verlag.

Devroye, L. 1986. *Non-Uniform Random Variate Generation.* New York: Springer-Verlag.

Gentle, J. E. 1998. *Random Number Generation and Monte Carlo Methods.* New York: Springer.

ed. P. Hellekalek and G. Larcher. 1998. *Random and Quasi-Random Point Sets.* volume 138 of *Lecture Notes in Statistics.* New York: Springer.

Hörmann, W. and J. Leydold. 2000. Automatic random variate generation for simulation input. In *Proceedings of the 2000 Winter Simulation Conference*, ed. J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, 675–682, Pistacaway, NJ. IEEE Press.

Knuth, D. E. 1998. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms.* Third ed. Reading, Mass.: Addison-Wesley.

Lagarias, J. C. 1993. Pseudorandom numbers. *Statistical Science*, **8**(1), 31–39.

Law, A. M and W. D. Kelton. 2000. *Simulation Modeling and Analysis.* Third ed. New York: McGraw-Hill.

L'Ecuyer, P. 1994. Uniform random number generation. *Annals of Operations Research*, **53**, 77–120.

L'Ecuyer, P. 1998. Random number generation. In *Handbook of Simulation*, ed. J. Banks, 93–137. Wiley. chapter 4.

L'Ecuyer, P. 1999a. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, **47**(1), 159–164.

L'Ecuyer, P. 1999b. Tables of maximally equidistributed combined LFSR generators. *Mathematics of Computation*, **68**(225), 261–269.

L'Ecuyer, P and T. H. Andres. 1997. A random number generator based on the combination of four LCGs. *Mathematics and Computers in Simulation*, **44**, 99–107.

L'Ecuyer, P. and R. Simard. 2000. On the performance of birthday spacings tests for certain families of random number generators. *Mathematics and Computers in Simulation.* To appear.

Marsaglia, G. 1985. A current view of random number generators. In *Computer Science and Statistics, Sixteenth Symposium on the Interface*, 3–10, North-Holland, Amsterdam. Elsevier Science Publishers.

Matsumoto, M and T. Nishimura. 1998. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, **8**(1), 3–30.

Niederreiter, H. 1992. *Random Number Generation and Quasi-Monte Carlo Methods*. volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*. Philadelphia: SIAM.

Tezuka, S. 1995. *Uniform Random Numbers: Theory and Practice*. Norwell, Mass.: Kluwer Academic Publishers.