TestU01: A C Library for Empirical Testing of Random Number Generators

PIERRE L'ECUYER and RICHARD SIMARD
Université de Montréal

We introduce *TestU01*, a software library implemented in the ANSI C language, and offering a collection of utilities for the empirical statistical testing of uniform random number generators (RNGs). It provides general implementations of the classical statistical tests for RNGs, as well as several others tests proposed in the literature, and some original ones. Predefined tests suites for sequences of uniform random numbers over the interval $(0, 1)$ and for bit sequences are available. Tools are also offered to perform systematic studies of the interaction between a specific test and the structure of the point sets produced by a given family of RNGs. That is, for a given kind of test and a given class of RNGs, to determine how large should be the sample size of the test, as a function of the generator's period length, before the generator starts to fail the test systematically. Finally, the library provides various types of generators implemented in generic form, as well as many specific generators proposed in the literature or found in widely-used software. The tests can be applied to instances of the generators predefined in the library, or to user-defined generators, or to streams of random numbers produced by any kind of device or stored in files. Besides introducing *TestU01*, the paper provides a survey and a classification of statistical tests for RNGs. It also applies batteries of tests to a long list of widely used RNGs.

Categories and Subject Descriptors: G.4 [**Mathematics of Computing**]: Mathematical Software; G.3 [**Mathematics of Computing**]: Probability and Statistics; Random Number Generation; Statistical software

General Terms: Statistical Software, Statistical Tests, Testing Random Number Generators

Additional Key Words and Phrases: Statistical Software, Random number generators, Random number tests

1. INTRODUCTION

Random numbers generators (RNGs) are needed for practically all kinds of computer applications, such as simulation of stochastic systems, numerical analysis, probabilistic algorithms, secure communications, computer games, and gambling machines, to name a few. The so-called random numbers may come from a physical device, like thermal noise from electronic diodes, but are more often the output of a small computer program which, from a given initial value called the *seed*, pro-

Authors' addresses: Pierre L'Ecuyer and Richard Simard, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, C.P. 6128, Succ. Centre-Ville, Montréal, H3C 3J7, Canada, e-mail: lecuyer@iro.umontreal.ca, simardr@iro.umontreal.ca.

duces a deterministic sequence of numbers that are supposed to *imitate* typical realizations of independent uniform random variables. The latter are sometimes called *pseudo-random number generators*, or *algorithmic* RNGs. Since they have a deterministic and periodic output, it is clear a priori that they do not produce independent random variables in the mathematical sense and that they cannot pass all possible statistical tests of uniformity and independence. But some of them have huge period lengths and turn out to behave quite well in statistical tests that can be applied in reasonable time. On the other hand, several popular RNGs, some available in commercial software, fail very simple tests [L'Ecuyer 2001].

Good RNGs are *not* designed by trying some arbitrary algorithms and applying empirical tests to the output until all the tests are passed. Instead, their design should involve a rigorous mathematical analysis of their period lengths and of the uniformity of the vectors of successive values that they produce over their entire period length; that's how independence is assessed theoretically [Knuth 1998; L'Ecuyer 2004]. However, once they have been selected and implemented, they must also be tested empirically. Statistical tests are also required for RNGs based (totally or partially) on physical devices. These RNGs are used to generate keys in cryptosystems and to generate numbers in lotteries and gambling machines, for example.

For a long time, the "standard" tests applied to RNGs were those described in earlier editions of the book of Knuth [1998]. Other tests, often more powerful to detect regularities in linear generators, were later proposed by Marsaglia [1985], Marsaglia [1996], and Marsaglia and Tsang [2002]. Some of these tests and new ones have been studied more extensively by Erdmann [1992], Marsaglia and Zaman [1993b], Vattulainen et al. [1995], L'Ecuyer and Simard [1999], L'Ecuyer et al. [2000], L'Ecuyer and Simard [2001], L'Ecuyer et al. [2002], Rukhin [2001], for instance. Pretty much all of them are available in *TestU01*.

Besides *TestU01*, the best-known public-domain statistical testing packages for RNGs are DIEHARD [Marsaglia 1996] and the test suite implemented by the National Institute of Standards and Technology (NIST) of the USA [Rukhin et al. 2001]. DIEHARD contains several statistical tests but has drawbacks and limitations. The sequence of tests as well as the parameters of these tests (sample size, etc.) are fixed in the package. The sample sizes are not very large: the entire test suite runs in a few seconds of CPU time on a standard desktop computer. As a result, they are not very stringent and the user has little flexibility for changing that. The package also requires that the random numbers to be tested are in a binary file in the form of 32-bit (exactly) integers. This file is to be passed to the testing procedures. This setup is quite restrictive. For instance, many RNGs produce numbers with less than 32 bits of accuracy (e.g., 31 bits is frequent) and DIEHARD does not admit that. The NIST package contains 15 tests, oriented primarily toward the testing and certification of RNGs used in cryptographic applications [Rukhin et al. 2001]. Another testing package worth mentioning is SPRNG [Mascagni and Srinivasan 2000], which implements the classical tests of Knuth [1998] plus a few others.

To summarize, empirical testing of RNGs is very important and yet, no comprehensive, flexible, state-of-the-art software is available for that, aside from the one

we are now introducing. The aim of the *TestU01* library is to provide a general and extensive set of software tools for statistical testing of RNGs. It implements a larger variety of tests than any other available competing library we know. It is also more flexible, the implementations are more efficient, and it can deal with larger sample sizes and a wider range of test parameters than for the other libraries. Tests are available for bit strings as well as for sequences of real numbers in the interval $(0, 1)$. It is easy to apply any of the tests (only) to specific bits from the output numbers, and to test decimated sequences (i.e., subsequences of non-successive output values at specified lags). The numbers can come from a file (binary or text) or can be produced by a program. Predefined test suites that contain sets of selected tests with fixed parameters are also available for users who prefer not to select the tests themselves. *TestU01* was developed and refined during the past 15 years and beta versions have been available over the Internet for a few years already. It will be maintained and updated on a regular basis in the future. It is available freely from the web page of the authors (currently at `http://www.iro.umontreal.ca/~simardr/testu01/tu01.html`).

The rest of this paper is organized as follows. In Sections 2 and 3, we review the (theoretical) quality criteria for good RNGs and their (empirical) testing. We then discuss the main empirical tests implemented in *TestU01*. The tests are classified in two categories: those that apply to a sequence of real numbers in $(0, 1)$ are examined in Section 4 and those designed primarily for a sequence of bits are covered in Section 5. In each category, the tests are further classified in subcategories. We discuss the links and similarities between different tests and identify specific types of regularities or defects that they are likely to detect. Section 6 gives an overview of the architecture of *TestU01* and provides an example of its use. In Section 7, we apply our main test batteries to a list of RNGs used in popular software or proposed in the literature. This is followed by a short conclusion.

2. QUALITY CRITERIA FOR RANDOM NUMBER GENERATORS

RNGs for all types of applications are designed so that their output sequence is a *good imitation* of a sequence of independent uniform random variables, usually over the real interval $(0, 1)$ or over the binary set $\{0, 1\}$. In the first case, the relevant *hypothesis* $\mathcal{H}_0^A$ to be tested is that the successive output values of the RNG, say $u_0, u_1, u_2, \ldots$, are independent random variables from the uniform distribution over the interval $(0, 1)$, i.e, i.i.d. $U(0, 1)$. In the second case, $\mathcal{H}_0^B$ says that we have a sequence of independent random bits, each taking the value 0 or 1 with equal probabilities independently of the others.

These two situations are strongly related, because under the i.i.d. $U(0, 1)$ hypothesis, any pre-specified sequence of bits (e.g., the bit sequence formed by taking all successive bits of u_0, or every second bit, or the first five bits of each u_i, etc.) must be a sequence of independent random bits. So statistical tests for bit sequences can be used as well (indirectly) for testing the null hypothesis $\mathcal{H}_0^A$.

In the $U(0, 1)$ case, $\mathcal{H}_0^A$ is equivalent to saying that for each integer $t > 0$, the vector $(u_0, \ldots, u_{t-1})$ is uniformly distributed over the t-dimensional unit cube $(0, 1)^t$. This cannot be true for algorithmic RNGs, because these vectors always take their values only from the *finite* set Ψ_t of all t-dimensional vectors of t successive values

that can be produced by the generator, from all its possible initial states (or seeds). The cardinality of this set cannot exceed the number of admissible seeds for the RNG. Assuming that the seed is chosen at random, vectors are actually generated in Ψ_t to approximate the uniform distribution over $(0,1)^t$. This suggests that Ψ_t should be very evenly distributed over the unit cube. Theoretical figures of merit for measuring this uniformity are discussed, e.g., in L'Ecuyer [2006], Niederreiter [1992], Tezuka [1995] and the references given there. They are used for the *design* of good RNGs. These criteria are much easier to compute for *linear* generators. This is one of the main reasons for the popularity of generators based on *linear recurrences*; e.g., linear congruential generators (LCGs), multiple recursive generators (MRGs), linear feedback shift-register (LFSR) generators, and generalized feedback shift-register (GFSR) generators [L'Ecuyer 1994; 2006; Tezuka 1995].

For a sequence of bits, the null hypothesis $\mathcal{H}_0^B$ cannot be formally true as soon as the length t of the sequence exceeds the number b of bits in the generator's state, because the number of distinct sequences of bits that can be produced cannot exceed 2^b. For $t > b$, the fraction of all 2^t sequences of t bits that can be visited is at most 2^{b-t}. The goal, then, is to make sure that those sequences that can be visited are "uniformly scattered" in the set of all 2^t possible sequences, and perhaps hard to distinguish.

Different quality criteria are used for RNGs in cryptology-related applications and for gambling machines in casinos. In these settings, an additional concern is *unpredictability* of the forthcoming numbers. The theoretical analysis of RNGs in cryptology is usually asymptotic, in the framework of computational complexity theory [Knuth 1998; Lagarias 1993; Goldreich 1999]. Nonlinear recurrences and/or output functions are used, which prevents one from measuring the uniformity of the set Ψ_t. As a result, empirical testing is even more necessary.

3. STATISTICAL TESTING

A *statistical test* for RNGs is defined by a *test statistic* Y, which is a function of a finite number of output real numbers u_n (or a finite number of bits, in the case of bit generators), whose distribution under $\mathcal{H}_0$ is known or can be closely approximated (here, $\mathcal{H}_0$ represents either $\mathcal{H}_0^A$ or $\mathcal{H}_0^B$). The number of different tests that can be defined is infinite and these different tests detect different problems with the RNGs. No universal test or battery of tests can guarantee, when passed, that a given generator is fully reliable for all kinds of simulations. But even if statistical tests can never *prove* that an RNG is foolproof, they can certainly improve our confidence in it. One can rightly argue that no RNG can pass every conceivable statistical test. The difference between the good and bad RNGs, in a nutshell, is that the bad ones fail very simple tests whereas the good ones fail only very complicated tests that are hard to figure out or impractical to run.

Ideally, when testing an RNG for simulation, Y should mimic the random variable of practical interest so that a bad structural interference between the RNG and the simulation problem will show up in the test. But this is rarely practical. This cannot be done, for example, for testing RNGs that are going to be used in general-purpose software packages. 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 standard statistical tests. The simple structure that permits some classes of generators to run very fast is often the source of major statistical weaknesses, which sometimes lead to totally wrong simulation results [Couture and L'Ecuyer 1994; Ferrenberg et al. 1992; L'Ecuyer and Simard 1999; L'Ecuyer et al. 2002; L'Ecuyer and Touzin 2004; Panneton and L'Ecuyer 2005; Tezuka et al. 1994]. Practical tools for detecting these deficiencies are needed. Offering a rich variety of empirical tests for doing that is the purpose of the *TestU01* library.

Classical statistical textbooks usually say that when applying a test of hypothesis, one must select beforehand a *rejection area* R whose probability under $\mathcal{H}_0$ equals the target test level (e.g., 0.05 or 0.01), and *reject* $\mathcal{H}_0$ if and only if $Y \in R$. This procedure might be appropriate when we have a fixed (often small) sample size, but we think it is *not* the best approach in the context of RNG testing. Indeed, when testing RNGs, the sample sizes are huge and can usually be increased at will. So instead of selecting a test level and a rejection area, we simply compute and report the *p-value* of the test, defined as

$$p = P[Y \geq y \mid \mathcal{H}_0]$$

where y is the value taken by the test statistic Y. If Y has a continuous distribution, then p is a $U(0,1)$ random variable under $\mathcal{H}_0$. For certain tests, this p can be viewed as a *measure of uniformity*, in the sense that it will be close to 1 if the generator produces its values with excessive uniformity, and close to 0 in the opposite situation.

If the p-value is extremely small (e.g., less than 10^{-10}), then it is clear that the RNG *fails* the test, whereas if it is not very close to 0 or 1, no problem is detected by this test. If the p-value is suspicious but does not clearly indicate rejection ($p = 0.002$, for example), then the test can be replicated "independently" with disjoint output sequences from the same generator until either failure becomes obvious or suspicion disappears. This approach is possible because there is usually no limit (other than CPU time) on the amount of data that can be produced by an RNG to increase the sample size or the number of replications of the test. When applying several tests to a given generator, p-values smaller than 0.01 or larger than 0.99 are often obtained by chance even if the RNG behaves correctly with respect to these tests (such values should normally appear approximately 2% of the time). In this case, suspicious values would not reappear systematically (unless we are extremely unlucky). Failure of a test typically depends on the structure of the point set Ψ_t and rarely on which part of the entire output sequence is tested (there are some exceptions; see, e.g., Panneton et al. 2006). Moreover, when a generator starts failing a test decisively, the p-value of the test usually converges to 0 or 1 exponentially fast as a function of the sample size of the test, when the sample size is increased further. Thus, suspicious p-values can easily be resolved by increasing the sample size.

In the case where Y has a *discrete distribution* under $\mathcal{H}_0$, we need to be more careful with the definition of p-value. In this case, we distinguish the *right p-value* $p_{\mathrm{R}} = P[Y \geq y \mid \mathcal{H}_0]$ and the *left p-value* $p_{\mathrm{L}} = P[Y \leq y \mid \mathcal{H}_0]$. We reject $\mathcal{H}_0$ when one of these two values is very close to 0. Why this distinction? Consider

for example a Poisson random variable Y with mean 1 under $\mathcal{H}_0$. If Y takes the value $y = 0$, the right p-value is $p_{\mathrm{R}} = P[Y \geq 0 \mid \mathcal{H}_0] = 1$. If we would use the same rejection procedure as in the case of continuous distributions (for which $p_{\mathrm{L}} = 1 - p_{\mathrm{R}}$), here we would reject $\mathcal{H}_0$ on the basis that the p-value is much too close to 1. However, $P[Y = 0 \mid \mathcal{H}_0] = 1/e \approx 0.368$, so it does not really make sense to reject $\mathcal{H}_0$ in this case. In fact, the left p-value here is $p_{\mathrm{L}} = 0.368$, so neither p_{L} nor p_{R} is close to 0. Note that we cannot define the left p-value as $p_{\mathrm{L}} = 1 - p_{\mathrm{R}} = P[Y < y \mid \mathcal{H}_0]$ in this case; in the example, this would give $p_{\mathrm{L}} = 0$.

Several authors have advocated and/or applied a *two-level* (or *second-order*) procedure for testing RNGs [Fishman 1996; Knuth 1998; L'Ecuyer 1992; Marsaglia 1985]. The idea is to generate N "independent" copies of Y, say $Y_1, \ldots, Y_N$, by replicating the first-order test N times on disjoint subsequences of the generator's output. Let F be the theoretical distribution function of Y under $\mathcal{H}_0$. If F is *continuous*, the transformed observations $U_1 = F(Y_1), \ldots, U_N = F(Y_N)$ are i.i.d. $U(0,1)$ random variables under $\mathcal{H}_0$. One way of performing the two-level test is to compare the empirical distribution of these U_j's to the uniform distribution, via a *goodness-of-fit* (GOF) test such as those of Kolmogorov-Smirnov, Anderson-Darling, Crámer-von Mises, etc.

If $U_{(1)}, \ldots, U_{(N)}$ are the N observations sorted by increasing order, the *Kolmogorov-Smirnov* (KS) test statistics D_N^+, D_N^-, and D_N are defined by

$$D_N^+ \;=\; \max_{1 \leq j \leq N} \left(j/N - U_{(j)} \right), \tag{1}$$

$$D_N^- \;=\; \max_{1 \leq j \leq N} \left(U_{(j)} - (j-1)/N \right), \tag{2}$$

$$D_N \;=\; \max\left(D_N^+, D_N^-\right), \tag{3}$$

the *Anderson-Darling* (AD) test statistic is

$$A_N^2 \;=\; -N - \frac{1}{N} \sum_{j=1}^{N} \left\{ (2j-1) \ln(U_{(j)}) + (2N+1-2j) \ln(1 - U_{(j)}) \right\}, \tag{4}$$

and the *Crámer-von Mises* (CVM) test statistic is

$$W_N^2 \;=\; \frac{1}{12N} + \sum_{j=1}^{N} \left(U_{(j)} - \frac{(j-0.5)}{N} \right)^2. \tag{5}$$

Their (approximate) distributions under $\mathcal{H}_0$ can be found in Anderson and Darling [1952], Darling [1960], Durbin [1973], Stephens [1970], Stephens [1986a], and Sinclair and Spurr [1988].

The p-value of the GOF test statistic is computed and $\mathcal{H}_0$ is rejected if this p-value is deemed too extreme, as usual. In *TestU01*, several of these GOF tests can actually be applied simultaneously for any given two-level test, if desired. This kind of flexibility could be convenient to study the power of these GOF tests for detecting the weaknesses of specific classes of generators.

Sometimes, the power of these tests can be increased by applying certain transformations to the sorted observations $U_{(1)}, \ldots, U_{(N)}$ before applying the test. A prominent example is the *spacings transformation*, defined as follows [Stephens 1986b]. We compute the spacings $S_i = U_{(i+1)} - U_{(i)}$ for $0 \leq i \leq N$, where $U_{(0)} = 0$,

$U_{(N+1)} = 1$, sort them by increasing order to obtain $S_{(0)} \le S_{(1)} \le \cdots \le S_{(N)}$, let $S_0 = (N+1)S_{(0)}$, and then compute $S_i = (N-i+1)(S_{(i)} - S_{(i-1)})$ and $V_i = S_0 + S_1 + \cdots + S_{i-1}$ for $i = 1, \ldots, N$. Under $\mathcal{H}_0$, $V_1, \ldots, V_N$ are distributed as N independent $U(0,1)$ random variables sorted by increasing order. This transformation is useful to detect *clustering*: If $U_{(i-1)}$ and $U_{(i)}$ are close to each other for several values of i, then several V_i's will be close to 0 and the AD test can detect it easily, whereas the standard GOF tests may not detect it on the original observations. Another example is the *power ratio transformation*, that also detects clustering [Stephens 1986b, Section 8.4]. It defines $U'_i = (U_{(i)}/U_{(i+1)})^i$ for $i = 1, \ldots, N$, and the V_i's are the U'_i sorted by increasing order. In *TestU01*, these transformations can be combined with any of the GOF tests mentioned above.

The arguments supporting the two-level tests are that (i) it sometimes permits one to apply the test with a larger total sample size to increase its power (for example, if the memory size of the computer limits the sample size of a single-level test), and (ii) it tests the RNG sequence at the local level, not only at the global level (i.e., there could be very bad behavior over short subsequences, which cancels out when averaging over larger subsequences). As an example of this, consider the extreme case of a generator whose output values are $i/2^{31}$, for $i = 1, 2, \ldots, 2^{31}-1$, in this order. A simple test of uniformity over the entire sequence would give a perfect fit, whereas the same test applied repeatedly over (disjoint) shorter sub-sequences would easily detect the anomaly.

In the case where the two-level test is motivated by (i) and not (ii), another way of performing the test at the second level is to simply add the N observations of the first level and reject $\mathcal{H}_0$ if the sum is too large or too small. For the great majority of the tests in *TestU01*, the distribution of Y is either chi-square, normal, or Poisson. In these three cases, the sum $\tilde{Y} = Y_1 + \cdots + Y_N$ has the same type of distribution. That is, if Y is chi-square with k degrees of freedom [resp., normal with mean μ and variance σ^2, Poisson with mean λ], $\tilde{Y}$ is chi-square with Nk degrees of freedom [resp., normal with mean $N\mu$ and variance $N\sigma^2$, Poisson with mean $N\lambda$]. In these situations, whenever a two-level test is performed, *TestU01* reports the results of the test based on $\tilde{Y}$ in addition to the second-order GOF tests specified by the user.

Our empirical investigations indicate that for a fixed total sample size Nn, when testing RNGs, a test with $N = 1$ is often more efficient than the corresponding test with $N > 1$. This means that for typical RNGs, the type of structure found in one reasonably long subsequence is often found in practically all subsequences of the same length. In other words, when an RNG started from a given seed fails spectacularly a certain test, it often fails that test for *most* admissible seeds. In the case where $N > 1$, the test based on $\tilde{Y}$ is usually more powerful than the second-order GOF tests that compare the empirical distribution of $F(Y_1), \ldots, F(Y_N)$ to the uniform. However, there are exceptions to these rules.

There are many tests for which the possible outcomes are partitioned in a finite number of categories; the test generates n such outcomes independently, counts how many fall in each category, and applies a chi-square test to compare these counts with the expected number in each category under $\mathcal{H}_0$. In this setting, we want to make sure that the expected number in each category is large enough ($\ge e_{\min}$ for some constant $e_{\min}$) for the chi-square test to be (approximately) valid. In

TestU01, whenever such a chi-square test occurs, adjacent categories are merged automatically until the expectation in each category is at least $e_{\min}$. The user does not have to care about this. The constant $e_{\min}$ is 10 by default but can be changed at any time.

In our implementation, this merging is done in a simple and naive way: before starting the tests, adjacent categories are merged until all the expected counts satisfy the constraint. Ryabko et al. [2004] and Ryabko et al. [2005] propose to do it with a two-phase adaptive approach instead. In the first phase (the *training phase*), one uses a very large number of categories and these categories are merged according to the empirical frequencies observed during that phase, divided by the theoretical (expected) frequencies. Categories with similar values of this ratio are merged to form the new categories for the *testing phase* (the second phase), in which the test is applied to a disjoint subsequence produced by the same generator. If the true expected frequencies for this generator differ from the theoretical ones and if they are well estimated by the empirical frequencies from the first phase, then this statistical-learning approach can boost significantly the power of the chi-square test. It also allows a very large number of categories in the first phase, much larger than $n\, e_{\min}$. However, this learning approach is not yet implemented in *TestU01*. We use other ways of dealing with very large numbers of categories (e.g., counting collisions).

There is an infinite number of possible tests of uniformity and independence for RNGs. Selection among them has to be somewhat subjective. However, there are natural choices that quickly come to mind. Certain tests are also motivated by important classes of applications.

What about comparing the *power* of these tests? This turns out to be a somewhat elusive issue, because the power of a test depends on the specific form of the alternative hypothesis, i.e., the specific type of structure that we want to detect. But when testing RNGs, there is no specified alternative; we would like to test against *anything* that departs from $\mathcal{H}_0$. We could select a specific class of RNGs and compare different tests in terms of their ability (power, efficiency) to detect some structure (non-randomness) in this specific class. The results will depend on the selected class of RNGs. It could be done for certain classes deemed important. *TestU01* is a good tool for those who want to study this empirically (see Section 6.4).

In the next two sections, we give an overview of the main empirical tests used in practice and implemented in *TestU01*. They are split in two categories: those that test $\mathcal{H}_0^A$ for sequence of real numbers in $(0, 1)$ and those that test $\mathcal{H}_0^B$ for a sequence of bits. Each category is further partitioned into subcategories.

4. TESTS FOR A SEQUENCE OF REAL NUMBERS IN $(0, 1)$

4.1 Tests on a single stream of n numbers

Measuring global uniformity. We want to test $\mathcal{H}_0^A$ for a sequence $u_1, \ldots, u_n$ in $(0, 1)$. One of the first ideas that come to mind is to compute the *empirical distribution* of $u_1, \ldots, u_n$ and compare it to the $U(0, 1)$ distribution, via the KS, AD, or other similar GOF tests. Even simpler is to compute the sample mean and sample variance, and compare them with the theoretical values 1/2 and 1/12. We can also

look at the sample autocorrelations of lags $j = 1, 2, \ldots$ and check if they are close enough to zero. Only very bad RNGs fail single-level versions of these simple tests. They measure the uniformity only at the *global* level, i.e., on average over the whole sequence of length n. Two-level versions with a large N and a small n may detect tendencies for the generator to get trapped for some time in areas of the state space where the mean or variance is smaller or larger than average, for example.

Measuring clustering. Another class of tests measure the *clustering* of the numbers $u_1, \ldots, u_n$. We sort these numbers by increasing order and compute the *overlapping m-spacings* $g_{m,i} = u_{(i+m)} - u_{(i)}$ between the sorted observations $u_{(1)}, \ldots, u_{(n)}$, where $u_{(0)} = 0$ and $u_{(n+i)} = 1 + u_{(i-1)}$ for $i > 0$. The test statistic has the general form $H^{(c)}_{m,n} = \sum_{i=0}^{n} h(n g_{m,i})$ where h is a smooth function such as $h(x) = x^2$ or $h(x) = \log(x)$ [L'Ecuyer 1997b]. This tests the empirical distribution of the u_i's in a different way than KS and AD tests, by testing if they tend to be more clustered than they should. For example, if the numbers are clustered by groups of about three to five that are very close to one another, the KS test is unlikely to detect it but a 3-spacings test will.

Run and gap tests. The run and gap tests [Knuth 1998] look more closely at the local anatomy of the sequence, trying to detect patterns. In the *gap test*, we select a set $A \subset [0, 1]$, with Lebesgue measure p (usually $A = [\alpha, \beta]$, an interval, in which case $p = \beta - \alpha$). The test counts the number of steps (or *gap size*) between any pair of successive visits to A. Let X_j be the number of gaps of size j, for $j \geq 0$. The test compares the frequencies $X_0, X_1, \ldots$ to their expected values under $\mathcal{H}_0^A$, via a chi-square test (after merging in a single class all values of j for which the expectation is too small). A generator whose visits to the set A tend to be clustered in time (the generator wanders in and out of A for some time, then goes away from A for a long while, and so on) will fail this test. The *run test* looks for another type of pattern; it collects the lengths of all increasing [or all decreasing] subsequences, counts how many there are of each length, and computes a modified chi-square statistic [Knuth 1998, page 67] based on these counts.

4.2 Tests based on n subsequences of length t

4.2.1 *Partitioning the unit hypercube and counting the hits per piece*

Recall that for an RNG that produces real numbers in $(0, 1)$, $\mathcal{H}_0^A$ is equivalent to saying that for every integers $n \geq 0$ and $t > 0$, the vector of output values $(u_n, \ldots, u_{n+t-1})$ has the uniform distribution over the t-dimensional unit hypercube $(0, 1)^t$. A natural approach to test this uniformity is the following. Partition the unit hypercube $(0, 1)^t$ into k pieces (cells) of volumes $p_0, \ldots, p_{k-1}$. Usually, $p_0 = \cdots = p_{k-1} = 1/k$, but not always. Then, generate n points $\mathbf{u}_i = (u_{ti}, \ldots, u_{ti+t-1}) \in (0, 1)^t$, for $i = 0, \ldots, n-1$, and count the number X_j of points falling in cell j, for $j = 0, \ldots, k-1$. Under $\mathcal{H}_0^A$, the vector $(X_0, \ldots, X_{k-1})$ has the *multinomial distribution* with parameters $(n, p_0, \ldots, p_{k-1})$. Any measure of distance (or *discrepancy*) between the numbers X_j and their expectations $\lambda_j = np_j$ can define a test statistic Y. We describe specific instances and variants of this type of test.

Serial tests. A simple way to partition the hypercube $(0,1)^t$ into $k = d^t$ subcubes (cells) of volume $1/k$ is by cutting the interval $(0,1)$ into d equal pieces for some integer $d > 1$. Then, $p_j = 1/k$ for all j. The tests that measure the overall discrepancy between the counts X_j and their expectation $\lambda = n/k$ are generally called *serial tests* of uniformity [Knuth 1998; L'Ecuyer et al. 2002].

In the original serial test [Good 1953; Knuth 1998], the distance to the exact distribution is measured by the chi-square test statistic

$$X^2 = \sum_{j=0}^{k-1} \frac{(X_j - \lambda)^2}{\lambda}$$

which has approximately a chi-square distribution with $k-1$ degrees of freedom when n/k is large enough. More generally, we may consider test statistics of the form

$$Y = \sum_{j=0}^{k-1} f_{n,k}(X_j)$$

where $f_{n,k}$ is a real-valued function which may depend on n and k. This class contains the *power divergence* statistics

$$D_\delta = \sum_{j=0}^{k-1} \frac{2}{\delta(1+\delta)} X_j \left[(X_j/\lambda)^\delta - 1\right], \tag{6}$$

studied in [Read and Cressie 1988; Wegenkittl 2001; 2002], where $\delta > -1$ is a real-valued parameter. We recover X^2 by taking $\delta = 1$. For $\delta = 0$, which means the limit as $\delta \to 0$, we obtain the *log-likelihood* statistic G^2 [Read and Cressie 1988]. It is equivalent to the *entropy* H, obtained by taking $f_{n,k}(x) = -(x/n)\log_2(x/n)$, via the relation $-H = G^2/(2n\ln 2) - \log_2(k)$. All these statistics have asymptotically the chi-square distribution with $k-1$ degrees of freedom when $n \to \infty$ for fixed k, under $\mathcal{H}_0^A$. An even more general class called the *generalized ϕ-divergence* statistics is introduced and studied in [Wegenkittl 1998; 2002], also in the asymptotic regime where k is fixed and $n \to \infty$. *TestU01* implements the power divergence class, but not this more general framework.

The value of δ that maximizes the power of the test in (6) depends on the alternative [Read and Cressie 1988; L'Ecuyer et al. 2002]. For example, suppose k_1 cells have probability $1/k_1$ and the other $k - k_1$ cells have probability 0. Then the power increases with δ if $k_1 \ll k$ (a *peak* in the distribution) and decreases with δ if $k - k_1 \ll k$ (a *hole*). When $k_1 \approx k/2$ (a *split*), the value of δ does not matter very much as this type of defect is easy to detect. A thin peak can be even easier if it contains sufficient probability mass, but a thin hole is extremely hard to detect.

For the classical serial test, it is usually recommended to have $n/k \geq 5$ for the chi-square test to be (approximately) valid. But this imposes a (practical) upper bound on the value of k, because k accumulators must be maintained in memory. Fortunately, this constraint is not really necessary. We say that we have a *sparse* serial test if $n/k \ll 1$ and a *dense* test if $n/k \gg 1$. In the sparse case, the distribution of D_δ under $\mathcal{H}_0^A$ is asymptotically normal when $n \to \infty$ and $n/k \to \lambda_0$ for some finite positive constant λ_0. The sparse tests tend to have more power than

the dense ones to detect typical defects of RNGs [L'Ecuyer et al. 2002], mainly because they permit larger values of k. When k is very large (e.g., $k \geq 2^{30}$ or more), using an array of k integers to collect the X_j's would use too much memory. In the sparse case, since most of the X_j's are zero, we use a hashing technique that maps the k values of j to a smaller number of memory locations.

Collisions, empty cells, and the like. In addition to the power divergence family of tests, *TestU01* supports other choices of $f_{n,k}$. For example, the number N_b of cells that contain exactly b points (for $b \geq 0$), the number W_b of cells that contain at least b points (for $b \geq 1$), and the number C of collisions, i.e., the number of times a point falls in a cell that already has one or more points in it. These statistics are related by $N_0 = k - W_1 = k - n + C$, $W_b = N_b + \cdots + N_n$, and $C = W_2 + \cdots + W_n$. They are studied and compared in L'Ecuyer et al. [2002], where good approximations for their distributions under $\mathcal{H}_0^A$ are given.

In each case, the test statistic is a measure of clustering: It decreases when the points are more evenly distributed between the cells. (For the entropy, this is true if we take $-H$.) The number of collisions C is often the most powerful test statistic among these to detect typical problems found in RNGs, mainly because it permits one to take $k \gg n$. For example, this test is very efficient to detect the peak and split alternatives mentioned earlier.

Under $\mathcal{H}_0^A$, C, W_2 and N_2 have asymptotically the Poisson distribution with mean μ_c where $n^2/(2k) \to \mu_c$ and the normal distribution with exact mean and variance that can be computed as explained in L'Ecuyer et al. [2002] if $n/k \to \lambda$, where $0 < \lambda < \infty$, and N_0 is asymptotically Poisson with mean $e^{-\gamma}$ when $n/k - \ln(k) \to \gamma$. Knuth [1998] gives an algorithm to compute the exact probabilities for C in $O(n \log n)$ time. However, in most cases of interest, these exact probabilities are very expensive to compute and are not needed, because the approximations based on the above asymptotics are excellent. For C, we use the Poisson approximation when $\lambda \leq 1$, the normal approximation when $\lambda > 1$ and $n > 10^5$, and the exact distribution otherwise.

Overlapping versions. There are also *overlapping* versions of the serial tests, where the points are defined by $\mathbf{u}_i = (u_i, \ldots, u_{i+t-1})$ for $i = 0, \ldots, n-1$. This has the advantage of providing more points (a larger n, and usually a more powerful test) for a given number of calls to the generator [L'Ecuyer et al. 2002]. Marsaglia and Zaman [1993b] name these overlapping serial tests *monkey tests*: they view the generator as a monkey typing "random" characters from a d-letter alphabet. The test counts how many times each t-letter word appears in the sequence typed by the monkey. They use the statistic N_0 and call the corresponding tests OPSO for $t = 2$, OTSO for $t = 3$, and OQSO for $t = 4$. When $d = 4$ and t around 10, they call it the DNA test.

In the overlapping case, the power divergence statistic D_δ no longer has an asymptotic chi-square distribution in the dense case. The analysis is more difficult because the successive cell numbers that are visited are no longer independent; they form a Markov chain instead. The relevant test statistic in the dense case is

$$\tilde{D}_{\delta,(t)} = D_{\delta,(t)} - D_{\delta,(t-1)}$$

where $D_{\delta,(t)}$ is the power divergence statistic (6) in t dimensions. Under $\mathcal{H}_0^A$, if

k is fixed and $n \to \infty$, $\tilde{D}_{\delta,(t)}$ is asymptotically chi-square with $d^t - d^{t-1}$ degrees of freedom. This was proved by Good [1953] for X^2 and generalized to $\tilde{D}_{\delta,(t)}$ by Wegenkittl [1998], Wegenkittl [2001] and L'Ecuyer et al. [2002].

In the sparse case, no good asymptotic results have been proved, as far as we know. Theorem 2 of Percus and Whitlock [1995] implies that $E[N_0] \approx ke^{-n/k}$ when $k \gg n \gg 0$, as in the non-overlapping case. Marsaglia and Zaman [1993b] suggest that N_0 is approximately normal with mean $ke^{-\lambda}$ and variance $ke^{-\lambda}(1 - 3e^{-\lambda})$, where $\lambda = n/k$, but this approximation is reasonably accurate only for $2 \le \lambda \le 5$ (approximately) and is very bad when λ is far outside this interval. Our simulation experiments indicate that C, W_2, and N_2 have approximately the same Poisson distribution as in the non-overlapping case when $\lambda \le 1$ (say) and n is large. We recommend using C with $\lambda \le 1$ and n as large as possible.

4.2.2 *Other ways of partitioning the unit hypercube*

Dividing the unit hypercube $(0,1)^t$ into $k = d^t$ sub-cubes of equal size as in the serial test is only one way of partitioning this hypercube into k pieces (or cells) and of mapping the vectors to the cell numbers. Another way, for example, is to find which of the $t!$ *permutations* of t objects would reorder the coordinates of $\mathbf{u}_i$ in increasing order. Number the $k = t!$ possible permutations from 0 to $k-1$, and let X_j be the number of vectors $\mathbf{u}_i$ that are reordered by permutation j, for $j = 0, \ldots, k-1$. The vector $(X_0, \ldots, X_{k-1})$ has again the multinomial distribution with parameters $(n, 1/k, \ldots, 1/k)$ and all the test statistics Y defined above can still be used and have the same distributions. This setting is appropriate, for instance, for testing an RNG that is going to be used to shuffle decks of cards for computer games or gambling machines. Then, the collision test applied to permutations could detect if certain permutations of the cards tend to appear more often than others. Knuth [1998] recommends using the chi-square statistic to test if all permutations are equally likely, but the number of collisions, C, is usually more sensitive and permits one to apply the test with a larger t, because we only need $n = O(\sqrt{k}) = O(\sqrt{t!})$ instead of $n \ge 5t!$.

The *argmax* test employs a different mapping: it defines X_j as the number of vectors $\mathbf{u}_i$ whose largest coordinate is the jth. This tests if the location of the maximum is evenly distributed among coordinates.

We can also compute the *sum, product, maximum*, etc., of $u_1, \ldots, u_t$, say Y, partition the range of possible values of Y in k intervals of equal probability under $\mathcal{H}_0^A$, and define X_j as the number of times Y falls in interval j. This gives a large family of multinomial tests. With a small or moderate t and a large n, these tests can detect clumping of small [or large] values in the sequence, for example.

The *poker test* of Kendall and Babington-Smith [1939], described by Knuth [1998], can be seen as yet another way of partitioning the hypercube. The test generates t integers in $\{0, \ldots, d-1\}$, looks at how many distinct integers there are, repeats this n times, and counts the frequencies of each number of distinct integers. For $t \le d$, this is equivalent to splitting $(0,1)^t$ into d^t sub-cubes and then regroup the sub-cubes into t classes as follows: take the point at the corner of the sub-cube closest to the origin and put the sub-cube in class j if this point has j distinct coordinates. These t classes form the final partition of $(0,1)^t$ used for the test. Note that the t cells do not have equal probabilities in this case.

4.2.3 *Alternatives to counting*

Focusing on a single cell. The CAT test mentioned by Marsaglia and Zaman [1993b] and analyzed by Percus and Whitlock [1995] (under the name of Monkey test) is a variation of the collision test where the collisions are counted in a single cell only. In its original version, the cells are determined as in the overlapping serial test, but the principle works without overlapping and for an arbitrary partition as well. Under $\mathcal{H}_0^A$, for large n, the number Y of points in the target cell has approximately the Poisson distribution with mean λ equal to the number of points generated multiplied by the volume of the target cell, assuming that the points are asymptotically pairwise independent when $n \to \infty$. When the points are independent (no overlap), Y has the binomial distribution. This test can be powerful only if the target cell happens to be visited too frequently, or if λ is large enough and the target cell is visited too rarely, due to a particular weakness of the generator.

Matsumoto and Kurita [1994, page 264] propose a very similar test with $t = 1$ but with two levels: generate n_0 numbers in $(0, 1)$ and count how many fall in a given interval $[\alpha, \beta]$, say Y. Repeat this n times and compare the distribution of the n realizations of Y to the binomial distribution with parameters n_0 and $p = \beta - \alpha$ by a chi-square test.

Time gaps between visits to cells. Instead of simply counting the visits to each cell of the partition, we can examine more detailed information; for instance collect the gaps (number of steps) between any two successive visits to a given cell, do this for each cell, and compute an appropriate function of all these gaps as a test statistic. This combines the ideas of the gap test and the multinomial test based on an hypercube partition. Maurer [1992] introduced a special case of this type of test and proposed as a test statistic the average of the logarithms of all the gaps, where the gaps start to be collected only after a long warm-up. The justification is that this average measures in some way the entropy of the sequence. He defined his test for bit strings (a block of bits determine the cell number) but the test applies more generally. In our experiments with this test several years ago, using a partition of the hypercube into sub-cubes, we found that it was always dominated by (i.e., was much less sensitive than) the sparse multinomial tests described earlier, and in particular by the collision test (at least for the sample sizes that we could handle and the examples that we tried). Wegenkittl [2001] and Hellekalek and Wegenkittl [2003] report similar results. Typically, several other tests fail long before this one does. Wegenkittl [2001] studies the connection between serial tests in the dense case and entropy estimators based on return times to cells (as in Maurer's test); he shows that the two are essentially equivalent. Since the serial test is usually not very sensitive in the dense case, it is no surprise then, that the same is true for Maurer's test.

Birthday spacings. An interesting refinement of the serial test is the *birthday spacings* test, which operates as follows [Marsaglia 1985; Knuth 1998; L'Ecuyer and Simard 2001]. We generate n points in k cells, just as for the serial test. Let $I_1 \le I_2 \le \cdots \le I_n$ be the cell numbers where these n points fall, sorted in increasing order. The test computes the spacings $I_{j+1} - I_j$, for $1 \le j < n$, and counts the number Y of collisions between these spacings. Under $\mathcal{H}_0^A$, Y is approximately

Poisson with mean $\lambda = n^3/(4k)$. If the test is replicated N times, the sum of the N values of Y (the total number of collisions) is computed and compared with the Poisson with mean $N\lambda$ to get the p-value. As a rule of thumb, the error of approximation of the exact distribution of the sum by this Poisson distribution is negligible when $Nn^3 \leq k^{5/4}$ [L'Ecuyer and Simard 2001]. This two-level procedure is usually more sensitive than applying a chi-square test to the collision counts, as proposed by Knuth [1998].

The rationale for this test is that certain types of RNGs have their points in cells that tend to be at regular spacings. This happens in particular for all LCGs when $t \geq 2$, because of their regular lattice structure in two or more dimensions [Knuth 1998; L'Ecuyer and Simard 2001]. To illustrate what happens, suppose that because of the regularity, all the spacings are multiples of some integer $b > 1$ and that they otherwise have the same distribution as if we had only $k' = k/b$ cells. Under this alternative, Y is approximately Poisson with mean $\lambda_1 = b\lambda$ instead of λ. If Y takes its most likely realization, $Y = b\lambda$, then the right p-value

$$p_{\mathrm{R}}(b\lambda) = P[Y \geq b\lambda \mid \mathcal{H}_0^A] \approx e^{-\lambda} \sum_{x=b\lambda}^{\infty} \lambda^x/x!$$

decreases exponentially in b. For $b = \lambda = 8$, for example, we already have $p_{\mathrm{R}}(64) = 1.9 \times 10^{-35}$. If $t \geq 2$ and the points are produced by an LCG with modulus near k, and if we take $n = (4k\lambda)^{1/3}$, we are in this situation, with a b that typically exceeds a few units. As a result, we can get a p-value below some fixed small threshold (e.g., $p_{\mathrm{R}}(y) < 10^{-15}$) with a sample size $n = O(k^{1/3})$. In other words, we have a clear failure of the test with a sample size n that increases as the *cubic root* of the modulus. Empirically, $n \approx 8k^{1/3}$ suffices to get $p_{\mathrm{R}}(y) < 10^{-15}$ for practically all LCGs with modulus near k. If the generator has a very bad lattice structure (all its points lie in only a few hyperplanes), then failure comes even faster than this. Knuth [1998, page 72] gives an example of this for a lagged-Fibonacci generator.

4.2.4 *Close pairs of points in space*

Again, we throw n points $\mathbf{u}_1, \ldots, \mathbf{u}_n$ independently and uniformly in the unit hypercube, but we now look at the distances between the points. This is studied by L'Ecuyer et al. [2000]. The distance is measured with the $\mathcal{L}_p$ norm $\|\cdot\|_p^o$ in the *unit torus* $(0,1)^t$, obtained by identifying (pairwise) the opposite sides of the unit hypercube, so that points that are face to face on opposite sides are "close" to each other. The distance between two points $\mathbf{u}_i$ and $\mathbf{u}_j$ is defined as $D_{n,i,j} = \|\mathbf{u}_j - \mathbf{u}_i\|_p^o$ where

$$\|\mathbf{x}\|_p^o = \begin{cases} \Big[\min(|x_1|, 1-|x_1|)^p + \ldots + \min(|x_t|, 1-|x_t|)^p\Big]^{1/p} & \text{if } 1 \leq p < \infty, \\ \max(\min(|x_1|, 1-|x_1|), \ldots, \min(|x_t|, 1-|x_t|)) & \text{if } p = \infty, \end{cases}$$

for $\mathbf{x} = (x_1, \ldots, x_t) \in [-1, 1]^t$. The reason for taking the torus is to get rid of the boundary effect: it is then much easier to obtain a good approximation of the relevant distributions (for the test statistics of interest) under $\mathcal{H}_0^A$.

Let $\lambda(n) = n(n-1)V_t(1)/2$, where $V_t(r)$ is the volume of the ball $\{x \in \mathbb{R}^t \mid \|x\|_p \leq r\}$. For each $\tau \geq 0$, let $Y_n(\tau)$ be the number of distinct pairs of points (X_i, X_j), with

$i < j$, such that $D_{n,i,j} \leq (\tau/\lambda(n))^{1/t}$. L'Ecuyer et al. [2000] show that under $\mathcal{H}_0^A$, for any fixed $\tau_1 > 0$ and large enough n, the truncated process $\{Y_n(\tau), 0 \leq \tau \leq \tau_1\}$ is approximately a Poisson process with unit rate. This implies that if $T_{n,0} = 0$ and $T_{n,1}, T_{n,2}, \ldots$ are the successive jump times of the process $\{Y_n\}$, then the spacings between these jump times are independent exponential random variables with mean 1, and thus for any fixed integer $m > 0$, the transformed spacings $W^*_{n,i} = 1 - \exp[-(T_{n,i} - T_{n,i-1})]$, for $i = 1, \ldots, m$, are approximately i.i.d. $U(0,1)$ if n is large enough. An *m-nearest-pairs* test compares the empirical distribution of these random variables to the uniform distribution. We use the Anderson-Darling test statistic for this comparison, because typically, when a generator has too much structure, the jump times of Y_n tend to cluster, so there tends to be several $W^*_{n,i}$'s near zero, and the Anderson-Darling test is particularly sensitive to detect that type of behavior [L'Ecuyer et al. 2000].

For a two-level test, there are several possibilities. A first one is to apply an AD test to the N (independent) p-values obtained at the first level. A second possibility is to pool all the Nm observations $W^*_{n,i}$ in a single sample and then apply an AD test of uniformity. These two possibilities are suggested by L'Ecuyer et al. [2000]. A third one is to superpose the N copies of the process Y_n, to obtain a process Y defined as the sum of the N copies of Y_n. We fix a constant $\tau_1 > 0$. Let J be the number of jumps of Y in the interval $[0, \tau_1]$ and let $T_1, \ldots, T_J$ be the sorted times of these jumps. Under $\mathcal{H}_0^A$, J is approximately Poisson with mean $N\tau_1$, and conditionally on J, the jump times T_j are distributed as J independent uniforms over $[0, \tau_1]$ sorted in increasing order. We can test this uniformity with an AD test on the J observations. It is also worthwhile to compare the realization of J with the Poisson distribution. We found several instances where the AD test of uniformity conditional on J was passed but the value of J was much too small. In one instance, we had $N\tau_1 = 100$ and observed $J = 2$. This is a clear failure based on the value of J. In yet another variant of the uniformity test conditional on J, we apply a "spacings" transformation to the uniforms before applying the AD test. This latter test is very powerful but is also very sensitive to the number of bits of "precision" in the output of the generator. For example, for $N = 20$, $n = 10^6$, $t = 2$, and $m = 20$, all generators returning less than 32 bits of precision fail the test.

Bickel and Breiman [1983] proposed a related goodness-of-fit test of uniformity based on the statistic $B_n = \sum_{i=1}^{n} (W_{n,(i)} - i/n)^2$, where $W_{n,(1)} \leq \ldots \leq W_{n,(n)}$ are the ordered values of the $W_{n,i} = 1 - \exp[-nV_t(D_{n,i})]$, $1 \leq i \leq n$, and $D_{n,i}$ is the distance from point i to its nearest neighbor (based on an arbitrary norm). These $W_{n,i}$ are approximately $U(0,1)$ under $\mathcal{H}_0^A$ and B_n measures their deviation from uniformity. In our extensive empirical experiments on RNG testing [L'Ecuyer et al. 2000], this test was less powerful than a two-level m-nearest-pairs test. It is also difficult to obtain an accurate approximation of the theoretical distribution of B_n.

4.3 Tests that generate n subsequences of random length

The tests in this category produce each subsequence by generating numbers u_j until some event happens, and the required number of u_j's is random. For example, in the *coupon collector's* test [Greenwood 1955; Knuth 1998], we partition the interval $(0,1)$ in d pieces of equal sizes and count how many random numbers u_j must

be generated to obtain at least one in each piece. The frequencies of the different counts are computed and compared with the theoretical frequencies by a chi-square test. This could also be defined with an arbitrary partition of $(0,1)^t$ into k pieces of equal volume.

The *sum collector* test [Ugrin-Sparac 1991] is similar, except that it counts how many uniforms are required until their sum exceeds a given constant.

5. TESTS FOR A SEQUENCE OF RANDOM BITS

Here we suppose that the generator produces a stream of bits $b_0, b_1, b_2, \ldots$, and we want to test the null hypothesis that the b_i are independent and take the values 0 or 1 with equal probability. These bits can come from any source. In particular, they can be extracted from a sequence of real numbers in $(0,1)$ by taking specific bits from each number. The standard procedure for doing this in *TestU01* is to take bits $r+1, \ldots, r+s$ from each number, i.e., skip the first r bits and take the s bits that follow, for some integers $r \geq 0$ and $s \geq 1$, and concatenate all these bits in a long string. The values of r and s are parameters of the tests.

5.1 One long binary stream of length n

Tests on binary sequences have been designed primarily in the area of cryptology, where high entropy and complexity are key requirements. The test of Maurer [1992], discussed earlier, is an entropy test, and the binary versions of the multinomial tests are essentially entropy tests as well [Wegenkittl 2001].

Linear complexity. One way of testing the complexity is to examine how the *linear complexity* L_ℓ of the first ℓ bits of the sequence increases as a function of ℓ. The linear complexity L_ℓ is defined as the smallest degree of a linear recurrence obeyed by the sequence. It is nondecreasing in ℓ and increases by integer-sized jumps at certain values of ℓ. In our implementation, L_ℓ is computed (updated) by the Berlekamp-Massey algorithm [Berlekamp 1984; Massey 1969] and requires $O(n^2 \log n)$ time overall. If the entire binary sequence follows a linear recurrence of order $k \ll n$ (this is the case for several widely-used classes of generators; see Section 7), then L_ℓ will stop increasing at $\ell = k$ and this would make the test fail. Carter [1989] and Erdmann [1992] examine two ways of testing the evolution of L_ℓ. The *jump complexity test* counts the number J of jumps in the linear complexity. Under $\mathcal{H}_0^B$ and for large n, J is approximately normally distributed with mean and variance given by [Carter 1989; Niederreiter 1991; Wang 1997]. The *jump size test* counts how many jumps of each size there are and compares these frequencies to the theoretical distribution (a geometric with parameter 1/2) by a chi-square test. A different type of test, used in the NIST suite [Rukhin et al. 2001], uses a two-level procedure with a large N and relatively smaller n. It computes N realizations of L_n, counts how many times each value has occurred, and uses a chi-square test to compare these counts to their theoretical expectations. In our empirical investigations, this test was strongly dominated by the jump complexity test and jump size test.

Lempel-Ziv complexity. Complexity can also be tested by counting the number W of distinct bit patterns that occur in the string. This number measures the compressibility of the sequence by the Lempel-Ziv compression algorithm given in

Ziv and Lempel [1978]. According to Kirschenhofer et al. [1994], under $\mathcal{H}_0^B$ and for large n, W is approximately normally distributed with mean $n/\log_2 n$ and variance $0.266\, n/(\log_2 n)^3$. However, these approximations of the mean and variance are not very accurate even for n as large as 2^{24}. Our implementation uses better approximations, obtained by simulation and confirmed with different types of (reliable) RNGs. This test turns out to be not very sensitive; generators that fail it tend to fail many other tests as well.

Fourier coefficients. Spectral tests on a binary sequence of $n = 2^k$ bits compute (some of) the *discrete Fourier coefficients*, which are complex numbers defined by

$$f_\ell = \sum_{j=0}^{n-1} (2b_j - 1) e^{2\pi \iota j \ell / n}, \qquad \ell = 0, 1, \ldots, n-1,$$

where $\iota = \sqrt{-1}$. Let $|f_\ell|$ be the modulus of f_ℓ. A first test, suggested in Rukhin et al. [2001], counts the number O_h of $|f_\ell|$'s that are smaller than some constant h, for $\ell \le n/2$. Under $\mathcal{H}_0^B$, for large n and $h = \sqrt{2.995732274 n}$, O_h is approximately normal with mean $\mu = 0.95n/2$ and variance $\sigma^2 = 0.05\mu$. Erdmann [1992] proposes several spectral-oriented tests and some are implemented in *TestU01*. However, the available approximations of the distributions of the test statistics under $\mathcal{H}_0^B$ are not very good.

Autocorrelations. The sample autocorrelation of lag ℓ in a bit sequence, defined as

$$A_\ell = \sum_{i=0}^{n-\ell-1} b_i \oplus b_{i+\ell},$$

where $\oplus$ is the exclusive-or operation (or addition modulo 2), also defines an interesting test statistic [Erdmann 1992]. Under $\mathcal{H}_0^B$, A_ℓ has the binomial distribution with parameters $(n - \ell, 1/2)$, which is approximately normal when $n - \ell$ is large.

Run and gap tests. The binary counterpart of the *run test* can be defined as follows. Every binary sequence has a run of 1's, followed by a run of 0's, followed by a run of 1's, and so on, or vice-versa if it starts with a 0. Suppose we collect the lengths of all runs of 1's and all runs of 0's until we have a total of $2n$ runs (n of each type). (The required length of the string to get $2n$ runs is random.) We count the number of runs of 1's of length j and the number of runs of 0's of length j, for $j = 1, \ldots, k$ for some integer k (regrouping the runs of length larger than k with those of length k) and apply a chi-square test on these $2k$ counts. Since any given run has length j with probability 2^{-j}, we readily know the expected number of runs of each length. Note that each run of 0's is a *gap* between the occurrence of 1's, and vice-versa, so this test can also be seen as a *gap test* for binary sequences.

5.2 Tests based on n bit strings of length m

5.2.1 *Partitioning the set of m-bit strings and counting hits in subsets*

We now consider tests that try to detect "dependencies" in bit strings of length m, in the sense that some of the 2^m possibilities are much more likely than others. All these tests use essentially (indirectly) the following pattern: regroup the 2^m

possibilities for the bit string into, say, k categories, count how many of the n strings fall in each category, and compare the results with the expectations, exactly as in the multinomial tests of Section 4.2.

Serial tests. The counterpart of the *serial test* for bit strings operates as follows: number the possible m-bit strings from 0 to $k-1 = 2^m - 1$ and let X_j be the number of occurrences of string j in the n strings. The same set of test statistics as in Section 4.2 can be used (chi-square, entropy, collisions, etc.). In the *overlapping* version, n bits are placed in a circle and each block of m successive bits on the circle determines a cell number. In this case, the test needs only n bits. The theoretical distribution is approximated as for an ordinary m-dimensional overlapping serial test with $d = 2$ and $t = m$.

The *CAT test* and the tests that collect the *gaps between visits* to states also have obvious binary versions, where the cells are replaced by the m-bit strings, constructed either with or without overlapping (the theoretical probabilities are different for the two cases). The test of Maurer [1992] discussed earlier was actually defined in this setting. Maurer [1992] proved that his test is *universal*, in the sense that it can detect any type of statistical defect in a binary sequence, when the test parameters and sample size go to infinity in the appropriate way (this result was generalized by Wegenkittl [2001]). In practice, however, it is typically less sensitive than the collision test for a comparable sample size.

Rank of a binary matrix. A powerful test to detect linear dependencies between blocks of bits is the *matrix rank* test [Marsaglia 1985; Marsaglia and Tsay 1985]: Fill up a $k \times \ell$ binary matrix row by row with a bit string of length $m = k \times \ell$, and compute the rank R of this binary matrix (the number of linearly independent rows). Repeat this n times and compare the empirical distribution of the n realizations of R with its theoretical distribution under $\mathcal{H}_0^B$, with a chi-square test. Bit sequences that follow linear recurrences fail this test for large enough m, because of the linear dependencies.

Longest run of 1's. The *longest head run* test [Földes 1979; Gordon et al. 1986] is a variant of the run test that looks at the length Y of the longest substring of successive 1's in a string of length m. This is repeated n times and the empirical distribution of the n realizations of Y is compared with its theoretical distribution by a chi-square test.

Hamming weights. To measure the clustering of 0's or of 1's in a bit sequence, we can examine the distribution of the *Hamming weights* of disjoint subsequences of length m. More clustering should lead to higher variance of the Hamming weights and to positive dependence between the Hamming weights of successive subsequences. The Hamming weights tests in *TestU01* generate n (non-overlapping) blocks of m bits and compute the Hamming weight of each block, say H_i for block i. Under $\mathcal{H}_0^B$, the H_i are i.i.d. binomial r.v.'s with parameters $(m, 1/2)$. Let X_j be the number of blocks having Hamming weight j, for $j = 0, \ldots, m$. A first test compares the distribution of the X_j's to their theoretical distribution (this test fits the framework of this section but the two that follow do not). A second test, used by Rukhin et al. [2001], computes the chi-square statistic $X^2 = (4/m)\sum_{i=1}^{n}(H_i - m/2)^2$, which

has approximately the chi-square distribution with n degrees of freedom under $\mathcal{H}_0^B$ if m is large enough. For $m = n$, this test degenerates to the *monobit* test [Rukhin et al. 2001], which simply counts the proportion of 1's in a string of n bits. A third test computes the linear correlation between the successive H_i's:

$$\hat{\rho}_1 = \frac{4}{(n-1)m} \sum_{i=1}^{n-1} (H_i - m/2)\,(H_{i+1} - m/2)\,.$$

Under $\mathcal{H}_0^B$ and for large n, $\hat{\rho}_1\sqrt{n-1}$ has approximately the standard normal distribution. This only tests *linear* dependence between Hamming weights. L'Ecuyer and Simard [1999] propose a different test of independence that takes $2n$ blocks of m bits. Again, H_i is the Hamming weight of block i. The pairs (H_i, H_{i+1}), for $i = 1, 3, \ldots, 2n-1$, can take $(m+1)^2$ possible values. The test counts the number of occurrences of each possibility and compare these counts to their theoretical expectations via a chi-square. This test showed significant dependence between the Hamming weights of successive output values of LCGs with special types of multipliers [L'Ecuyer and Simard 1999].

Random walk tests. From a bit sequence of length ℓ, we can define a random walk over the integers as follows: the walk starts at 0 and at step j, it moves by 1 to the left if $b_j = 0$, and by 1 to the right if $b_j = 1$. If we define $S_0 = 0$ and $S_k = \sum_{j=1}^{k}(2b_j - 1)$ for $k > 0$, then the process $\{S_k,\, k \geq 0\}$ is this random walk. Under $\mathcal{H}_0^B$, we have (from the binomial distribution)

$$p_{k,y} \stackrel{\text{def}}{=} P[S_k = y] = 2^{-k} \binom{k}{(k+y)/2} \qquad \text{if } k+y \text{ is even}$$

and $p_{k,y} = 0$ otherwise. In what follows, we assume that ℓ is even. We define the statistics:

$$\begin{aligned}
H &= \ell/2 + S_\ell/2, \\
M &= \max\{S_k,\ \ 0 \leq k \leq \ell\}\,, \\
J &= 2\sum_{k=1}^{\ell/2} \mathbb{I}[S_{2k-1} > 0], \\
P_y &= \min\{k : S_k = y\} \qquad \text{for } y > 0, \\
R &= \sum_{k=1}^{\ell} \mathbb{I}[S_k = 0], \\
C &= \sum_{k=3}^{\ell} \mathbb{I}[S_{k-2}S_k < 0],
\end{aligned}$$

where $\mathbb{I}$ denotes the indicator function. Here, H is the number of steps to the right, M is the maximum value reached by the walk, J is the fraction of time spent on the right of the origin, P_y is the first passage time at y, R is the number of returns to 0, and C is the number of sign changes. The theoretical probabilities for these

statistics under $\mathcal{H}_0^B$ are as follows [Feller 1968]:

$$
\begin{aligned}
P[H = k] &= P[S_\ell = 2k - \ell] = p_{\ell,2k-\ell} = 2^{-\ell}\binom{\ell}{k}, \quad 0 \le k \le \ell, \\
P[M = y] &= p_{\ell,y} + p_{\ell,y+1}, \quad 0 \le y \le \ell, \\
P[J = k] &= p_{k,0}\, p_{\ell-k,0}, \quad 0 \le k \le \ell,\ k \text{ even}, \\
P[P_y = k] &= (y/k)p_{k,y}, \\
P[R = y] &= p_{\ell-y,y}, \quad 0 \le y \le \ell/2, \\
P[C = y] &= 2p_{\ell-1,2y+1}, \quad 0 \le y \le (\ell-1)/2.
\end{aligned}
$$

The n-block test of Vattulainen et al. [1995] is equivalent to counting the proportion of walks for which $H \ge \ell/2$. Takashima [1996] applies some of these random walk tests to LFSR generators.

An omnibus random walk test implemented in *TestU01* takes two even integers $m > m_0 > 0$ as parameters and generate n random walks of length m. For each ℓ in $\{m_0, m_0 + 2, \ldots, m\}$, the test computes the n values of the statistics $H, \ldots, C$ defined above, and compares their empirical distributions with the corresponding theoretical ones via a chi-square test.

We have a different implementation with $m = m_0$ that first applies a linear transformation to the original bit sequence: For a fixed vector of bits $(c_0, \ldots, c_{t-1})$, not all zero, the new bit j becomes $y_j = c_0 b_j + \ldots + c_{t-1} b_{j+t-1}$. This requires $m + t - 1$ bits in the original sequence. At step j, the walk goes left if $y_j = 0$ and goes right if $y_j = 1$.

5.2.2 *Close pairs*

We have the following analogue of the close pair test for binary sequences. Each of the n points in t dimensions is determined by a bit string of length $m = st$, where each coordinate has s bits. Often, each such block of s bits would be obtained by making one call to a generator and extracting s bits from the output value. The distance between two points X_i and X_j is defined as $2^{-b_{i,j}}$, where $b_{i,j}$ is the maximal value of b such that the first b bits in the binary expansion of each coordinate are the same for both X_i and X_j. This means that if the unit hypercube is partitioned into 2^{tb} cubic boxes by dividing each axis into 2^b equal parts, the two points are in the same box for $b \le b_{i,j}$, but they are in different boxes for $b > b_{i,j}$. Let $D = \min_{1 \le i < j \le n} 2^{-b_{i,j}}$ be the minimal distance between any two points. For a given pair of points, $P[b_{i,j} \ge b] = 2^{-tb}$. One has $D \le 2^{-b}$ if and only if $-\log_2 D = \max_{1 \le i < j \le n} b_{i,j} \ge b$, if and only if at least b bits agree for at least one pair, and the probability that this happens is approximately

$$
q_b \stackrel{\text{def}}{=} P[D \le 2^{-b}] \approx 1 - \left(1 - 2^{-tb}\right)^{n(n-1)/2}.
$$

If exactly $b = -\log_2 D$ bits agree, the left and right p-values are $p_{\mathrm{L}} = q_b$ and $p_{\mathrm{R}} = 1 - q_{b-1}$, respectively. If $N > 1$, the two-level test computes the minimum of the N copies of D and uses it as a test statistic. The p-value is obtained from $P[\min\{D_1, D_2, \ldots, D_N\} \le 2^{-b}] \approx 1 - (1 - q_b)^N$.

Certain types of LFSR generators are constructed so that if we partition $(0,1)^t$ into $2^{t\ell}$ sub-cubes of equal size, where $2^{t\ell}$ is the cardinality of the set Ψ_t, then each

sub-cube contains exactly one point from Ψ_t [L'Ecuyer 1996b; 1999b; Tezuka 1995]. For these generators, D has the lower bound $2^{-\ell}$, so they systematically fail the test if n is large enough.

6. ORGANIZATION OF *TESTU01*

The software tools of *TestU01* are organized in four classes of modules: those implementing RNGs, those implementing statistical tests, those implementing predefined batteries of tests, and those implementing tools for applying tests to entire families of generators. The names of the modules in those four classes start with the letters `u`, `s`, `b`, and `f`, respectively, and we shall refer to them as the `u`, `s`, `b`, and `f` modules. The name of every public identifier (type, variable, function, ...) is prefixed by the name of the module to which it belongs.

6.1 Generator implementations: the u modules

TestU01 contains a large selection of predefined uniform RNGs. Some are good and many are bad. These implementations are provided for experimental purpose only. They permit one to perform empirical studies with several types of empirical tests and RNGs. Several classes of generators are implemented in generic form; the user provides the parameters together with the seed, at construction time. Other RNGs have fixed hard-coded parameters, exactly as in their originally proposed versions.

Generator objects. The `unif01` module provides the basic tools for defining and manipulating uniform RNGs. It contains the (structured) type `unif01_Gen`, which implements the definition of an arbitrary RNG object. Every RNG in *TestU01* is an object of this type. The `unif01_Gen` structure contains the name of the generator, its parameters, its state, a function that can write the state in readable format, a function `GetU01` that returns a $U(0,1)$ random variate and advances the generator by one step, and a function `GetBits` that returns a block of 32 bits and advances the generator by one step. Statistical tests in *TestU01* can be applied only to objects of this type. To test a new generator not already available in *TestU01*, it may be implemented as a `unif01_Gen` object, normally by implementing all fields of `unif01_Gen`. Simplified constructors are also provided that only require an implementation of *either* `GetU01` or `GetBits`. For the situations where the random numbers to be tested are already in a file, there are predefined special types of `unif01_Gen` objects that simply read the numbers from a file, either in text or in binary format. The `GetU01` and `GetBits` methods simply return these numbers in the appropriate format. This is convenient, for instance, if the generator to be tested cannot easily be called directly in C, or if the numbers are produced by a physical device, or if they are already in a file for some other reason.

Output filters. When a statistical test is applied to a generator, the test usually invokes the `GetU01` and `GetBits` methods only *indirectly*, through the basic *filters* `unif01_StripD(g, r)`, `unif01_StripL(g, r, d)`, and `unif01_StripB(g, r, s)`. These three filters get the next output value u from the generator `g`, drop the r most significant bits and shift the other ones to the left by r positions, and return the resulting uniform $u' = 2^r u \bmod 1$, the integer $\lfloor du' \rfloor$ in $\{0, \ldots, d-1\}$, and the integer $\lfloor 2^s u' \rfloor$ (a block of s bits), respectively.

Other types of output filters predefined in *TestU01* can be inserted between the RNG and these basic filters. These filters are implemented as *containers*: each filter applied to a `unif01_Gen` object g is itself a `unif01_Gen` object g' that contains g in one of its fields. Thus, several filters can be applied in series to *any* given generator.

A *Double* filter, for example, is a generator g' that calls the contained generator g twice to get u_1 and u_2, and returns $u = (u_1 + hu_2) \bmod 1$, each time it has to produce a $U(0,1)$ random number, where h is a fixed constant (usually a negative power of 2). This permits one to increase the number of bits of precision in the generator's output. A *Trunc* filter does essentially the opposite. It calls the contained generator and truncates the output to its s most significant bits, for a selected integer s. This is useful to study empirically how sensitive are some tests to the number of bits of precision in the output values. A *Bias* filter introduces a deliberate bias in the output; it generates a random variate of constant density with probability p over the interval $[0, a)$, and of constant density with probability $1-p$ over the interval $[a, 1)$, where a and p are selected constants, by inversion from u. A *Lux* filter implements luxury on an arbitrary generator: out of every group of L random numbers, it keeps the first k and skips the next $L-k$. This is done in order to remove possible long-term correlations in a stream of random numbers. A *Lac* filter is a generator g' that picks up only selected output values from g, and discards the other ones. The indices of the selected values are specified by a set of integers $0 \le i_1 < i_2 < \cdots < i_k$. If the output sequence of g is $u_0, u_1, u_2, \ldots$, then the output sequence of g' will be

$$u_{i_0}, u_{i_1}, \ldots, u_{i_{k-1}}, u_{L+i_0}, u_{L+i_1}, \ldots, u_{L+i_{k-1}}, u_{2L+i_0}, u_{2L+i_1}, \ldots$$

For example, if $k = 3$ and $I = \{0, 3, 5\}$, the output sequence will be the numbers $u_0, u_3, u_5, u_6, u_9, u_{11}, u_{12}, \ldots$ To obtain the decimated sequence $u_{s-1}, u_{2s-1}, u_{3s-1}, \ldots$, one should take $k = 1$ and $I = \{s-1\}$. Taking $k = 2$ and $I = \{0, s\}$ will return $u_0, u_s, u_{s+1}, u_{2s+1}, \ldots$ Additional filters can be defined by the users if needed.

Parallel generators. Suppose we want to build a sequence by taking output values from k generators in a round-robin fashion, say L numbers at a time from each generator, and apply statistical tests to the resulting sequence. This is easily achieved via the *parallel* filter. It suffices to pass the k generators and the value of L when creating the filter and it will correspond to a generator that produces the required sequence. The k generators can also be k different substreams from the same underlying generator, started with different seeds. This tool is convenient for testing the dependence between the substreams for a random number generator with multiple streams [L'Ecuyer et al. 2002] or between the different generators for parallel simulation.

Combinations of generators. Predefined tools permit one to combine two or three generators of *any types* either by adding their outputs modulo 1 or by a bit-wise exclusive-or of their outputs. The combined generator is implemented as a `unif01_Gen` object that contains its components, which are also `unif01_Gen` objects. For example, if g combines g_1 and g_2 by addition modulo 1, then each output value of g will be produced by output values u_1 and u_2 from g_1 and g_2, and returning $u = (u_1 + u_2) \bmod 1$. These combination tools give a lot of room for experimentation. The users can define other ones if needed.

Predefined generators. Nearly 200 different generators are predefined and implemented in *TestU01*, including LCGs, MRGs, combined MRGs, lagged-Fibonacci generators, add-with-carry, subtract-with-borrow, and multiply-with-carry generators, LFSR and combined LFSR generators, GFSR and twisted GFSR generators, Mersenne twisters, WELL generators, different families of nonlinear inversive generators (explicit and implicit), quadratic and cubic congruential generators, and Weyl and nested Weyl generators, and so on. Specific generators taken from popular software such as Unix, Excel, Visual Basic, Java, Mathematica, Matlab, Maple, S-Plus, etc., or proposed in articles, are also available.

6.2 Statistical tests: the s modules

The statistical tests are implemented in several `s` modules. When invoking a function that applies a test to a generator g, we pass the generator object as the first parameter, then a pointer to a structure `res` that will contain detailed information about what happened during the test (if we just want a printout and not recover these detailed results, we just pass `NULL` for this parameter), and then the parameters of the test. Most tests have the following three parameters in common: N is the number of (second-level) replications, n is the sample size at the first level, and r is the number of (most significant) bits that are stripped from the output before applying the test. The level of detail in the printout of the test results can be selected by the user. The structure `res` is convenient in case we want to do further computations with the results of the tests or with the contents of intermediate collectors.

6.3 Batteries of tests: the b module

Some users of *TestU01* may not have a clear idea of which tests they want to apply to their generators and what parameter values they should use for the tests. Many prefer predefined suites (or batteries) of statistical tests, with fixed parameters, that they can apply with a single function call. The DIEHARD and NIST batteries, for example, are popular mainly for that reason. A good battery should contain different types of tests that can detect weaknesses of different kinds. We need small batteries than can run in a few seconds as well as more stringent ones that may require several hours. The small batteries may be used to detect gross defects in generators or errors in their implementation.

Six predefined batteries of tests are available in *TestU01*; three of them are for sequences of $U(0,1)$ random numbers and the three others are for bit sequences. In the first category, we have *SmallCrush*, *Crush*, and *BigCrush*, whose respective running times to test a generator such as MT19937 (for example), on a computer with an AMD Athlon 64 processor running at 2.4 GHz, are 14 seconds, 1 hour, and 5.5 hours. To test an RNG, it is recommended to start with the quick battery *SmallCrush*. If everything is fine, one could try *Crush*, and finally the more time-consuming *BigCrush*. Note that some of the tests compute more than one statistic (and p-value). In the current version, *Crush* uses approximately 2^{35} random numbers and applies 96 statistical tests (it computes a total of 144 test statistics and p-values), whereas *BigCrush* uses approximately 2^{38} random numbers and applies 106 tests (it computes 160 test statistics and p-values). The tests are those described in sections 4 and 5, plus a few others. This includes the classical tests

described in Knuth [1998], e.g. the *run, poker, coupon collector, gap, max-of-t* and *permutation* tests. There are *collision* and *birthday spacings* tests in 2, 3, 4, 7, 8 dimensions, several *close pairs* tests in 2, 3, 5, 7, 9 dimensions, and *correlation* tests. Some tests use the generated numbers as a sequence of "random" bits: *random walk* tests, *linear complexity* tests, a *Lempel-Ziv compression* test, several *Hamming weights* tests, *matrix rank* tests, *run* and *correlation* tests, among others.

The batteries *Rabbit*, *Alphabit* and *BlockAlphabit* are for binary sequences (e.g., a cryptographic pseudorandom generator or a source of random bits produced by a physical device). They were originally designed to test a finite sequence contained in a binary file. When invoking the battery, one must specify the number n_{B} of bits available for each test. When the bits are in a file, n_{B} must not exceed the number of bits in the file, and each test will reuse the same sequence of bits starting from the beginning of the file (so the tests are not independent). When the bits are produced by a generator, each test uses a different stream. In both cases, the parameters of each test are chosen automatically as a function of n_{B}. *Rabbit* and *Alphabit* apply 38 and 17 different statistical tests, respectively. *BlockAlphabit* applies the *Alphabit* battery of tests repeatedly to a generator or a binary file after reordering the bits by blocks of different sizes (with sizes of $2, 4, 8, 16, 32$ bits). On a 64-bit AMD Athlon processor running at 2.4 GHz, *Rabbit* takes about 2 seconds to test a stream of 2^{20} bits in a file and about 9 minutes for a stream of 2^{30} bits. *Alphabit* takes less than 1 second for 2^{20} bits and about 1 minute for 2^{30} bits.

We do not claim that the tests selected in the predefined batteries are independent or exhaustive. We tried to avoid selecting tests that are obviously nearly equivalent. But beyond that, we think it is quite difficult to measure the independence of the proposed tests and to compare their efficiencies for testing RNGs, because there is no fixed alternative hypothesis, as explained earlier. We also kept a few tests that have always been dominated by others in our experiments (e.g., Maurer's test, Lempel-Ziv complexity, etc.), mainly because of their popularity. These tests may turn out to be useful in the future, against certain alternatives that we have not met yet; otherwise we may eventually remove them from the batteries.

6.4 Testing families of generators: the f modules

The `f` modules provide tools designed to perform systematic studies of the interaction between certain types of tests and the structure of the point sets produced by given families of RNGs. The idea is to find prediction formulas for the sample size n_0 for which the test starts to reject an RNG decisively, as a function of its period length ρ. For each family, an RNG of period length near 2^i has been pre-selected, on the basis on some theoretical criteria that depend on the family, for all integers i in some interval (from 10 to 30, for example). There are several families such as LCGs (single or combined), MRGs of order 2 and 3, LFSRs (single or combined), cubic and combined cubic generators, and inversive generators (explicit and implicit).

In experiments already performed with specific classes of generators and tests [L'Ecuyer and Hellekalek 1998; L'Ecuyer et al. 2000; L'Ecuyer and Simard 2001], the results were often surprisingly regular, in the sense that a regression model of the form $\log n_0 = a \log \rho + \delta$, where a is a constant and δ a small noise, fits very well. Typically, for a given RNG that fails a test, as the sample size of the test is increased, the p-value remains "reasonable" for a while, say for n up to some

threshold n_0, and then converges to 0 or 1 exponentially fast as a function of n. The result gives an idea of what period length ρ of the RNG is required, within a given family, to be safe with respect to the considered test, for a given sample size n.

For full-period LCGs with good spectral test behavior for example, the relationships $n_0 \approx 16\,\rho^{1/2}$ for the collision test and $n_0 \approx 16\,\rho^{1/3}$ for the birthday spacings test have been obtained. This means that no LCG is safe with respect to these particular tests unless its period length ρ is so large that generating $\rho^{1/3}$ numbers is practically unfeasible. A period length of 2^{48} or less, for example, does not satisfy this requirement.

6.5 An example

Figure 1 illustrates the use of various facilities in *TestU01*. In this C program, the function `xorshift` implements a 32-bit xorshift generator [Marsaglia 2003]. These generators with only three shifts have been shown to have deep defects [Panneton and L'Ecuyer 2005] and *SmallCrush* detects that unequivocally. After putting this generator inside a wrapper to make it compatible with *TestU01* by invoking `CreateExternGenBits`, the program applies the battery *SmallCrush* to it.

In the next block of instructions, we set up a subtract-with-borrow (SWB) generator `gen2`, based on the recurrence $x_i = (x_{i-8} - x_{i-48} - c_{i-1}) \bmod 2^{31}$ and $u_i = x_i/2^{31}$. The array `A` contains the initial values (seed) for this generator and it is filled using the first 31 bits of 48 successive output values of `gen1`. We then apply a `Lac` filter to `gen2`, with *lacunary indices* $\{0, 40, 48\}$, to create another generator `gen3` (the filtered generator). The output sequence of `gen3`, in terms of the original sequence numbering of `gen2`, is $u_0, u_{40}, u_{48}, u_{49}, u_{89}, u_{97}, u_{98}$, and so on. With these particular lacunary indices, it turns out that all the points produced by three successive output values of `gen3` fall (with up to 31 bits of accuracy) in exactly two planes in the unit hypercube [Couture and L'Ecuyer 1994]. For this reason, `gen3` fails several simple tests in three dimensions, including the small *birthday spacings* test (with $t = 3$, $k = 2^{12t}$, and sample size $n = 10000$) applied here.

Next, the program creates the `LFSR113` generator of L'Ecuyer [1999b] in `gen1` and the `MRG31k3p` generator of L'Ecuyer and Touzin [2000] in `gen2`, then invokes `CreateCombAdd2` to create a generator `gen3` which adds the outputs of `gen1` and `gen2` modulo 1, and finally `CreateDoubleGen` produces a fourth generator `gen4` whose output values are formed by adding two output values of `gen3` modulo 1, after dividing one of them by 2^{24}. This provides an output with a full 53 bits of accuracy. The battery *Crush* is then applied to `gen4`. Finally, all generators are deleted and the program ends.

The detailed output of this program is not given here (to save space), but the results can be summarized as follows. The xorshift generator `gen1` fails the *SmallCrush* battery with four p-values smaller than 10^{-10}, `gen3` fails the birthday spacings test with a p-value smaller than 10^{-300}, and `gen4` passes all the tests of *Crush*.

7. TESTING WIDELY USED RNGS

We applied the test suites *SmallCrush*, *Crush* and *BigCrush* to a long list of well-known or widely used RNGs. Table I gives a representative subset of the results. The RNGs not in the table behave in a similar way as other RNGs shown in the

```
#include "TestU01.h"
#include <stddef.h>
#define IMAX 48
#define LACN 3
static unsigned int y = 2463534242U;

unsigned int xorshift (void) {
   y ^= (y << 13);   y ^= (y >> 17);   return y ^= (y << 5);
}

int main (void) {
   unif01_Gen *gen1, *gen2, *gen3, *gen4;
   unsigned long A[IMAX + 1];
   long Lac[LACN] = {0, 40, 48};
   int i;
   gen1 = unif01_CreateExternGenBits ("Xorshift-32", xorshift);
   bbattery_SmallCrush (gen1);

   for (i = 0; i < IMAX; i++)
      A[i] = unif01_StripB (gen1, 0, 31);
   gen2 = ucarry_CreateSWB (8, 48, 0, 2147483648, A);
   gen3 = unif01_CreateLacGen (gen2, LACN, Lac);
   smarsa_BirthdaySpacings (gen3, NULL, 1, 10000, 0, 4096, 3, 1);
   unif01_DeleteLacGen (gen3);   ucarry_DeleteSWB (gen2);
   unif01_DeleteExternGen01 (gen1);

   gen1 = ulec_Createlfsr113 (12345, 12345, 12345, 12345);
   gen2 = ulec_CreateMRG31k3p (123, 123, 123, 123, 123, 123);
   gen3 = unif01_CreateCombAdd2 (gen1, gen2, "Comb_ulec");
   gen4 = unif01_CreateDoubleGen (gen3, 24);
   bbattery_Crush (gen4);
   unif01_DeleteDoubleGen (gen4);   unif01_DeleteCombGen (gen3);
   ulec_DeleteGen (gen2);   ulec_DeleteGen (gen1);
   return 0;
}
```

Fig. 1. Example of a C program using *TestU01*.

table.

For each RNG, the column $\log_2 \rho$ gives the logarithm in base 2 of the period length ρ (when we know it). The columns t-32 and t-64 give the CPU time (in seconds) required to generate 10^8 random numbers on a 32-bit computer with an Intel Pentium processor of clock speed 2.8 GHz and a 64-bit computer with an AMD Athlon 64 processor of clock speed 2.4 GHz, respectively, both running Red Hat Linux. We must emphasize that our implementations of specific RNGs are not necessarily the fastest possible, but most should be close to the best available. The missing entries are the generators for which we do not have an implementation; we tested them by direct calls to the relevant software (e.g., Mathematica). In the following columns, for each test suite and each RNG, we give the number of statistical tests for which the p-value is *outside* the interval $[10^{-10},\ 1-10^{-10}]$. Although the choice of this interval is arbitrary, this can be interpreted conservatively as the *number of clear failures*. A blank entry means that no p-value was outside that in-

terval, whereas a long dash (—) indicates that the test suite was not applied to this particular RNG (usually, because the RNG was already failing decisively a smaller battery). Some entries have a second number in parentheses; this is the number of tests whose p-value was in $(10^{-10}, 10^{-4}] \cup [1 - 10^{-4}, 1 - 10^{-10})$. These can be viewed as *suspect* p-values. For instance, when applying *Crush* to the generator LCG($2^{31} - 1$, $2^{15} - 2^{10}$, 0), we obtained 59 clear failures and 7 additional suspect p-values. The generators in the table are regrouped by categories according to their underlying algorithm.

LCGs. The first category contains linear congruential generators, which obey the recurrence $x_i = (ax_{i-1} + c) \bmod m$ with output function $u_i = x_i/m$; they are denoted by LCG(m, a, c). LCGs with power-of-two modulus $m = 2^e$ are known to be badly behaved, especially in their least significant bits [L'Ecuyer 1990]. This is confirmed by the tests. The first one, `LCG`(2^{24}, 16598013, 12820163), is the toy generator used in Microsoft VisualBasic 6.0. `LCG`(2^{31}, 65539, 0) is the infamous RANDU [IBM 1968]. `LCG`(2^{32}, 69069, 1) is from Marsaglia [1972] and has been much used in the past, alone and in combination with other RNGs. `LCG`(2^{32}, 1099087573, 0) is an LCG with "optimal multiplier" found by Fishman [1990]. `LCG`(2^{46}, 5^{13}, 0) was used by the Numerical Aerodynamic Simulation at NASA Ames Research Center in their benchmarks [Agarwal et al. 2002]. `LCG`(2^{48}, 25214903917, 11) is the `drand48` from the Unix standard library. The RNG in `java.util.Random` of the Java standard library is based on the same recurrence, but uses two successive values to build each output, via $u_i = (2^{27}\lfloor x_{2i}/2^{22} \rfloor + \lfloor x_{2i+1}/2^{21} \rfloor)/2^{53}$. This is an improvement over `drand48`, but still not acceptable. `LCG`(2^{48}, 5^{19}, 0) has been the traditional MCNP generator used at the Los Alamos National Laboratory [Brown and Nagaya 2002]. `LCG`(2^{48}, 33952834046453, 0) is one of the LCGs with "optimal multipliers" found by Fishman [1990]; it is used in LAPACK. `LCG`(2^{48}, 44485709377909, 0) was used on CRAY systems and is provided as an intrinsic function on the IBM XLF and XLHPF Fortran compilers as well as in PESSL (Parallel Engineering and Scientific Subroutine Library) from IBM. `LCG`(2^{59}, 13^{13}, 0) is the basic generator in the NAG mathematical library [NAG 2002] and is available in VSL, the Vector Statistical Library from the Intel Math Kernel Library [Intel 2003]. `LCG`(2^{63}, 5^{19}, 0) and `LCG`(2^{63}, 9219741426499971445, 0) are recommanded for future use at the Los Alamos National Laboratory [Brown and Nagaya 2002].

The following LCGs have a prime modulus m, which makes them a little better behaved than those with a power-of-two modulus, but they are still unacceptable because their period is too short and they fail several tests. `LCG`($2^{31} - 1$, 16807, 0) was proposed long ago by Lewis et al. [1969] and has been very popular in books and software for many years. `LCG`($2^{31} - 1$, 742938285, 0) and `LCG`($2^{31} - 1$, 950706376, 0) use two of the "optimal multipliers" found by Fishman and Moore III [1986]. This last as well as `LCG`(2^{31}–1, 397204094, 0) are available amongst others in IMSL [1997]. `LCG`($10^{12} - 11$, 427419669081, 0) was used in Maple 9.5 [Maplesoft 2006] and in MuPAD 3 [SciFace Software 2004]. This LCG has been replaced by `MT19937` (see below) in Maple 10. LCGs with multipliers of the form $a = \pm 2^r \pm 2^s$, such as `LCG`($2^{31} - 1$, $2^{15} - 2^{10}$, 0) and `LCG`($2^{61} - 1$, $2^{30} - 2^{19}$, 0), were proposed by Wu [1997]. L'Ecuyer and Simard [1999] have shown that they are statistically weaker than LCGs with the same m and a good multiplier a, because they do not mix the

Table I. Results of test batteries applied to well-known RNGs

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush		Crush		BigCrush	
LCG(2^{24}, 16598013, 12820163)	24	3.9	0.66	14		—		—	
LCG(2^{31}, 65539, 0)	29	3.3	0.65	14		125	(6)	—	
LCG(2^{32}, 69069, 1)	32	3.2	0.67	11	(2)	106	(2)	—	
LCG(2^{32}, 1099087573, 0)	30	3.2	0.66	13		110	(4)	—	
LCG(2^{46}, 5^{13}, 0)	44	4.2	0.75	5		38	(2)	—	
LCG(2^{48}, 25214903917, 11)	48	4.1	0.65	4		21	(1)	—	
Java.util.Random	47	6.3	0.76	1		9	(3)	21	(1)
LCG(2^{48}, 5^{19}, 0)	46	4.1	0.65	4		21	(2)	—	
LCG(2^{48}, 33952834046453, 0)	46	4.1	0.66	5		24	(5)	—	
LCG(2^{48}, 44485709377909, 0)	46	4.1	0.65	5		24	(5)	—	
LCG(2^{59}, 13^{13}, 0)	57	4.2	0.76	1		10	(1)	17	(5)
LCG(2^{63}, 5^{19}, 1)	63	4.2	0.75			5		8	
LCG(2^{63}, 9219741426499971445, 1)	63	4.2	0.75			5	(1)	7	(2)
LCG(2^{31}–1, 16807, 0)	31	3.8	3.6	3		42	(9)	—	
LCG(2^{31}–1, $2^{15} - 2^{10}$, 0)	31	3.8	1.7	8		59	(7)	—	
LCG(2^{31}–1, 397204094, 0)	31	19.0	4.0	2		38	(4)	—	
LCG(2^{31}–1, 742938285, 0)	31	19.0	4.0	2		42	(5)	—	
LCG(2^{31}–1, 950706376, 0)	31	20.0	4.0	2		42	(4)	—	
LCG(10^{12}–11, 427419669081, 0)	39.9	87.0	25.0	1		22	(2)	34	(1)
LCG(2^{61}–1, $2^{30} - 2^{19}$, 0)	61	71.0	4.2			1	(4)	3	(1)
Wichmann-Hill	42.7	10.0	11.2	1		12	(3)	22	(8)
CombLec88	61	7.0	1.2			1			
Knuth(38)	56	7.9	7.4			1	(1)	2	
ran2	61	7.5	2.5						
CLCG4	121	12.0	5.0						
Knuth(39)	62	81.0	43.3				(1)	3	(2)
MRGk5-93	155	6.5	2.0						
DengLin (2^{31}–1, 2, 46338)	62	6.7	15.3		(1)	11	(1)	19	(2)
DengLin (2^{31}–1, 4, 22093)	124	6.7	14.6		(1)	2		4	(2)
DX-47-3	1457	—	1.4						
DX-1597-2-7	49507	—	1.4						
Marsa-LFIB4	287	3.4	0.8						
CombMRG96	185	9.4	2.0						
MRG31k3p	185	7.3	2.0				(1)		
MRG32k3a	191	10.0	2.1						
MRG63k3a	377	—	4.3						
LFib(2^{31}, 55, 24, +)	85	3.8	1.1	2		9		14	(5)
LFib(2^{31}, 55, 24, −)	85	3.9	1.5	2		11		19	
ran3		2.2	0.9		(1)	11	(1)	17	(2)
LFib(2^{48}, 607, 273, +)	638	2.4	1.4			2		2	
Unix-random-32	37	4.7	1.6	5	(2)	101	(3)	—	
Unix-random-64	45	4.7	1.5	4	(1)	57	(6)	—	
Unix-random-128	61	4.7	1.5	2		13		19	(3)
Unix-random-256	93	4.7	1.5	1	(1)	8		11	(1)

Generator	$\log_2 \rho$	t-32	t-64	SmallCrush		Crush		BigCrush	
Knuth-ran_array2	129	5.0	2.6			3		4	
Knuth-ranf_array2	129	11.0	4.5						
SWB(2^{24}, 10, 24)	567	9.4	3.4	2		30		46	(2)
SWB(2^{24}, 10, 24)[24, 48]	566	18.0	7.0			6	(1)	16	(1)
SWB(2^{24}, 10, 24)[24, 97]	565	32.0	12.0						
SWB(2^{24}, 10, 24)[24, 389]	567	117.0	43.0						
SWB(2^{32}–5, 22, 43)	1376	3.9	1.5		(1)	8		17	
SWB(2^{31}, 8, 48)	1480	4.4	1.5		(2)	8	(2)	11	
Mathematica-SWB	1479	—	—	1	(2)	15	(3)	—	
SWB(2^{32}, 222, 237)	7578	3.7	0.9			2		5	(2)
GFSR(250, 103)	250	3.6	0.9	1		8		14	(4)
GFSR(521, 32)	521	3.2	0.8			7		8	
GFSR(607, 273)	607	4.0	1.0			8		8	
Ziff98	9689	3.2	0.8			6		6	
T800	800	3.9	1.1	1		25	(4)	—	
TT800	800	4.0	1.1			12	(4)	14	(3)
MT19937	19937	4.3	1.6			2		2	
WELL1024a	1024	4.0	1.1			4		4	
WELL19937a	19937	4.3	1.3			2	(1)	2	
LFSR113	113	4.0	1.0			6		6	
LFSR258	258	6.0	1.2			6		6	
Marsa-xor32 (13, 17, 5)	32	3.2	0.7	5		59	(10)	—	
Marsa-xor64 (13, 7, 17)	64	4.0	0.8	1		8	(1)	7	
Matlab-rand	1492	27.0	8.4			5		8	(1)
Matlab-LCG-Xor	64	3.7	0.8			3		5	(1)
SuperDuper-73	62	3.3	0.8	1	(1)	25	(3)	—	
SuperDuper64	128	5.9	1.0						
R-MultiCarry	60	3.9	0.8	2	(1)	40	(4)	—	
KISS93	95	3.8	0.9			1		1	
KISS99	123	4.0	1.1						
Brent-xor4096s	131072	3.9	1.1						
ICG(2^{31}–1, 1, 1)	31	58.0	69.0			6		11	(6)
ICG(2^{31}–1, 22211, 11926380)	31	74.0	69.0			5		10	(8)
EICG(2^{31}–1, 1, 1)	31	49.0	57.0			6		13	(7)
EICG(2^{31}–1, 1288490188, 1)	31	55.0	64.0			6		14	(6)
SNWeyl	32	12.0	4.2	1		56	(12)	—	
Coveyou-32	30	3.5	0.7	12		89	(5)	—	
Coveyou-64	62	—	0.8			1		2	
LFib(2^{64}, 17, 5, $*$)	78	—	1.1						
LFib(2^{64}, 55, 24, $*$)	116	—	1.0						
LFib(2^{64}, 607, 273, $*$)	668	—	0.9						
LFib(2^{64}, 1279, 861, $*$)	1340	—	0.9						
ISAAC		3.7	1.3						
AES (OFB)		10.8	5.8						
AES (CTR)	130	10.3	5.4						(1)
AES (KTR)	130	10.2	5.2						
SHA-1 (OFB)		65.9	22.4						
SHA-1 (CTR)	442	30.9	10.0						

bits well enough.

All these LCGs with moduli up to 2^{61} fail several tests and should be discarded.

Combined LCGs. A combined LCG provides an efficient way of implementing an LCG with a large composite modulus, equal to the product of the moduli of the components. The combined LCGs of Wichmann and Hill [1982] is of that form, with three components of moduli near 30300, whose outputs are added modulo 1. This is the RNG provided in Microsoft Excel since 2003. L'Ecuyer [1988] introduced a slightly different type of combination, giving a generator that can be closely approximated by an LCG. He proposed `CombLec88`, a combination of two LCGs of moduli near 2^{31}, used in several software packages, including RANLIB, CERNLIB, Boost, Octave, and Scilab. Knuth [1998, page 108], Eq. (38), proposed a variant of it named `Knuth(38)` in the table. It can be found in the GNU Scientific Library (GSL) [Galassi et al. 2004], under the name `fishman2x` (although G. Fishman never proposed this generator). Press and Teukolsky [1992] proposed a slightly modified version of `CombLec88` (with shuffled output) named `ran2`. L'Ecuyer and Andres [1997] also proposed `CLCG4`, a combination of four LCGs, which is definitely more robust but also slower. `CLCG4` and `ran2` are the only ones that pass all the tests; this is not surprising because the other combined LCGs are essentially equivalent to LCGs with moduli smaller than 2^{61}. `CombLec88` fails a four-dimensional birthday spacings test in Crush, with a p-value of around 10^{-98} (the number of collisions is about twice the expected number).

MRGs and combined MRGs. A multiple recursive generator (MRG) is defined by a recurrence of the form $x_i = (a_1 x_{i-1} + \cdots + a_k x_{i-k}) \bmod m$, with output function $u_i = x_i/m$. In the table, `Knuth(39)` is from Knuth [1998, page 108], Eq. (39), and has recurrence $x_i = (271828183\, x_{i-1} + 314159269\, x_{i-2}) \bmod (2^{31} - 1)$. `MRGk5-93` is from [L'Ecuyer et al. 1993] and has recurrence $x_i = (107374182\, x_{i-1} + 104480\, x_{i-5}) \bmod (2^{31} - 1)$. The `DengLin`$(m, k, a)$ have the form $x_i = (-x_{i-1} + a x_{i-k}) \bmod m$ and were proposed by Deng and Lin [2000]. The DX$*$ are selected among the long-period MRGs proposed by Deng [2005]. The recurrences are $x_i = (2^{26} + 2^{19})(x_{i-1} + x_{i-24} + x_{i-47}) \bmod (2^{31} - 1)$ for `DX-47-3` and $x_i = (-2^{25} - 2^7)(x_{i-7} + x_{i-1597}) \bmod (2^{31} - 1)$ for `DX-1597-2-7`. `Marsa-LFIB4` is an MRG with recurrence $x_i = (x_{i-55} + x_{i-119} + x_{i-179} + x_{i-256}) \bmod 2^{32}$ proposed by Marsaglia [1999].

`CombMRG96`, `MRG32k3a`, and `MRG31k3p` are combined MRGs for 32-bit computers, proposed by L'Ecuyer [1996a], L'Ecuyer [1999a], and L'Ecuyer and Touzin [2000], respectively. `MRG63k3a` is a combined MRG for 64-bit computers, proposed by L'Ecuyer [1999a]. `MRG32k3a` is now widely used in simulation and statistical software such as VSL, SAS, Arena, Automod, Witness, and SSJ. All these MRGs except the smaller ones perform very well in the tests. `MRG31k3p` came up with a suspect p-value of 7.3×10^{-5} for a random walk test. To see if this generator was really failing this test, we repeated the same test five times on disjoint subsequences and obtained the following p-values: 0.66, 0.86, 0.34, 0.10, 0.22. We conclude that the suspect p-value was a statistical fluke.

Decimation. For certain types of recurrences of order k, the vectors of the form $(x_i, \ldots, x_{i+k})$ have a bad structure, because of the excessive simplicity of the re-

currence. For example, if $x_i = (x_{i-r} + x_{i-k}) \bmod m$, then all nonzero vectors of the form (x_{i-k}, x_{i-r}, x_i) lie in only two planes! For this reason, Lüscher [1994] and other authors proposed to use a *decimated* version of the sequence: from each block of ℓ successive terms x_i from the recurrence, retain the first c to produce c output values and discard the other ones. Often, c is fixed and the parameter ℓ is called the *luxury level*. When this type of decimation is used, we add $[c, \ell]$ after the generator's name, in the table.

Lagged-Fibonacci. Lagged Fibonacci generators LFib$(m, r, k, \texttt{op})$ use the recurrence $x_i = (x_{i-r}\ \texttt{op}\ x_{i-k}) \bmod m$, where `op` is an operation that can be + (addition), − (subtraction), ∗ (multiplication), ⊕ (bitwise exclusive-or). With the + or − operation, these generators are in fact MRGs. `LFib`(2^{31}, 55, 24, +) was recommended in Knuth [1981] based on a suggestion by G. J. Mitchell and D. P. Moore [unpublished]. `LFib`(2^{31}, 55, 24, −) is `Ran055` in Matpack [Gammel 2005]. The `ran3` generator of Press and Teukolsky [1992] is essentially a `LFib`(10^9, 55, 24, −). LFib(2^{48}, 607, 273, +) is one of several lagged Fibonacci RNGs included in the Boost Random Number Library [Maurer et al. 2004]. The `Unix-random`'s are RNGs of different sizes derived from BSD Unix and available as function `random()` on several Unix-Linux platforms. They are `LFib`($2^{32}, 7, 3, +$), `LFib`($2^{32}, 15, 1, +$), `LFib`($2^{32}, 31, 3, +$) and `LFib`($2^{32}, 63, 1, +$), with the least significant bit of each random number dropped. `Knuth-ran_array2` and `Knuth-ranf_array2` are `LFib`(2^{30}, 100, 37, −)[100, 1009] and `LFib`(2^{30}, 100, 37, +)[100, 1009], respectively, proposed by Knuth [1998]. The second version is proposed in the most recent reprints of his book, since 2002, as an improvement to the first. It is the only one in this class that passes the tests.

Subtract with borrow. These generators, denoted SWB(m, r, k), employ a linear recurrence like a subtractive lagged-Fibonacci but with a borrow [Marsaglia and Zaman 1991]: $x_i = (x_{i-r} - x_{i-k} - c_{i-1}) \bmod m$ where $c_i = \lfloor (x_{i-r} - x_{i-k} - c_{i-1})/m \rfloor$, and $u_i = x_i/m$. It is now well-known that they are approximately equivalent to LCGs with large moduli and bad structure, similar to that of additive or subtractive lagged-Fibonacci RNGs [Couture and L'Ecuyer 1994; 1997; L'Ecuyer 1997a]. SWB(2^{24}, 10, 24), SWB($2^{32} - 5$, 22, 43), and SWB(2^{31}, 8, 48) were proposed by Marsaglia and Zaman [1991]. SWB(2^{24}, 10, 24)$[24, \ell]$, called RANLUX, was advocated in [Lüscher 1994; James 1994] with luxury levels $\ell = 24, 48, 97, 223, 389$. In its original form it returns only 24-bit output values. For our tests, we use a version with 48 bits of precision obtained by adding two successive numbers modulo 1 as $u_i = (x_{2i}/2^{24} + x_{2i+1}/2^{48}) \bmod 1$. This makes the RNG much more robust, but also slower. SWB($2^{32} - 5$, 22, 43) was proposed as a component of the RNG of Marsaglia et al. [1990]. `Mathematica-SWB` is implemented in the `Random[]` function of Mathematica, version 5.2 and earlier. It uses two steps of the recurrence SWB(2^{31}, 8, 48) to produce real-valued output in (0,1) with 53 bits of precision, apparently via $u_i = (x_{2i}/2^{31} + x_{2i+1})/2^{31}$. SWB($2^{32}$, 222, 237) was proposed by Marsaglia [1999]. All these generators fail several tests unless we use decimation with a large-enough luxury level, which slows them down considerably.

LFSR and GFSR generators. These generators are all based on linear recurrences modulo 2. If $u_{i,j}$ denotes the jth bit of the output u_i, then $\{u_{i,j},\ i \geq 0\}$ follows

the same binary linear recurrence for all j. All these generators obviously fail statistical tests that measure the linear complexity of the binary sequences or that try to detect linear dependence (for example, the matrix-rank test). The well-known trinomial-based GFSRs, with recurrences of the form $x_i = x_{i-r} \oplus x_{i-k}$ (we use the notation GFSR(k, r)), are in fact lagged-Fibonacci RNGs with the bitwise *exclusive or* operation $\oplus$.

`GFSR`(250, 103) is the $R250$ of Kirkpatrick and Stoll [1981]. It has been widely used in the past and is included in VSL. `GFSR`(521, 32) has also been very popular; we use the initialization of Ripley [1990]. `GFSR`(607, 273) was proposed by Tootill et al. [1973]. `Ziff98` is a pentanomial-based GFSR from Ziff [1998] with recurrence $x_i = x_{i-471} \oplus x_{i-1586} \oplus x_{i-6988} \oplus x_{i-9689}$.

`T800` and `TT800` are twisted GFSR generators proposed in [Matsumoto and Kurita 1992; 1994]. `MT19937` is the famous Mersenne twister of Matsumoto and Nishimura [1998]. It is used in Goose, SPSS, EViews, GNU R, VSL and in many other software packages. The WELL generators were introduced by Panneton et al. [2006] to correct some weaknesses in `MT19937`. `LFSR113` and `LFSR258` are the combined Tausworthe generators of L'Ecuyer [1999b] designed specially for 32-bit and 64-bit computers, respectively. `MT19937` and `WELL19937a` fail only two linear complexity tests, with p-values larger than $1 - 10^{-15}$ (all LFSR and GFSR-type generators fail these two tests). `WELL19937a` also has a slightly suspect p-value of 7.2×10^{-5} for a Hamming independance test in *Crush*.

The xorshift RNGs are a special form of LFSR generators proposed by Marsaglia [2003], and further analyzed by Brent [2004] and Panneton and L'Ecuyer [2005]. In a *xorshift* operation, a word (block of bits) is xored (bitwise exclusive-or) with a shifted (either left or right) copy of itself. Marsaglia [2003] recommends the 3-shift RNGs `Marsa-xor32(13, 17, 5)` for 32-bit computers, and `Marsa-xor64(13, 7, 17)` for 64-bit computers, either alone or in combination with some other RNGs.

All these generators fail linear complexity tests for bit sequences, because their bit sequences obey linear recurrences by construction. Some also fail several other tests.

Mixed combined generators. `Matlab-rand` is the uniform generator `rand` included in MATLAB [Moler 2004]. It is a XOR combination of a SWB(2^{53}, 12, 27) with `Marsa-xor32(13, 17, 5)` mentioned above. MATLAB uses another uniform RNG to generate its normal variates: it is the same xorshift RNG combined with LCG(2^{32}, 69069, 1234567) by addition modulo 2^{32}. `SuperDuper-73` is the Super-Duper generator from Marsaglia et al. [1973]: it is a XOR combination of the LCG(2^{32}, 69069, 0) with a small shift-register RNG. `SuperDuper-73` is used in S-PLUS [Ripley and Venables 1994; MathSoft Inc. 2000]. `SuperDuper64` is a 64-bit additive combination of an LCG with an LFSR, proposed by Marsaglia [2002]. `R-MultiCarry` was proposed by Marsaglia [1997] and concatenates two 16-bit multiply-with-carry generators; it is used in the R statistical package [Project 2003]. `KISS93`, from Marsaglia and Zaman [1993a], combines an LCG with two small LFSR generators. `KISS99`, from Marsaglia [1999], combines an LCG, a 3-shift LFSR, and two multiply-with-carry generators. `Brent-xor4096s`, proposed by Brent [2004], combines a xorshift RNG with 4 xorshifts with a Weyl generator. Brent [2004] proposes several other RNGs of that type.

Some of these mixed combined generators pass the tests, but many do not. `KISS93` fails a linear complexity test based on bit 30, with a p-value larger than $1 - 10^{-15}$, in both *Crush* and *BigCrush*.

Inversive generators. These RNGs use nonlinear recurrences. The inversive congruential generator ICG (m, a, b) uses the recurrence $x_{j+1} = (a\overline{x}_j + b) \bmod m$, where $x\overline{x} = 1 \bmod m$ if $x \neq 0$, and $\overline{0} = 0$ [Eichenauer and Lehn 1986]. The output is $u_j = x_j/m$. ICG($2^{31} - 1$, 1, 1) was suggested by Eichenauer-Herrmann [1992]. ICG($2^{31} - 1$, 22211, 11926380) was suggested by Hellekalek [1995]. The explicit inversive congruential generator EICG (m, a, b) uses the recurrence $x_j = \overline{(a\,j + b)} \bmod m$ with output $u_j = x_j/m$. EICG($2^{31} - 1$, 1, 1) and EICG($2^{31} - 1$, 1288490188, 1) were suggested by [Hellekalek 1995; 1998]. A significant drawback of inversive RNGs is that they require modular inversion, a slow and costly operation. All the inversive generators tested here have a short period and fail the tests.

Other nonlinear generators. `SNWeyl` is a shuffled version of a nested Weyl generator defined by $\nu_j = m(j(j\alpha \bmod 1) \bmod 1) + 1/2$, $u_j = \nu_j(\nu_j\alpha \bmod 1) \bmod 1$, where $m = 1234567$ and $\alpha = \sqrt{2} \bmod 1$ [Holian et al. 1994; Liang and Whitlock 2001]. Without the shuffling, the generator is much worse. These RNGs have been used in statistical physics. They are not portable, since their output depends on the processor and the compiler, and they also fail the tests. `Coveyou-32` and `Coveyou-64` [Coveyou 1969] are quadratic generators with recurrence $x_{j+1} = x_j(x_j + 1) \bmod 2^\ell$, with $\ell = 32$ and 64 respectively. They both fail, although the larger one is better behaved.

The multiplicative lagged Fibonacci generators `LFib`(2^{64}, 17, 5, $*$), `LFib`(2^{64}, 55, 24, $*$), `LFib`(2^{64}, 607, 273, $*$) and `LFib`(2^{64}, 1279, 861, $*$) are from Marsaglia [1985] and are available in SPRNG [Mascagni and Srinivasan 2000]. These generators pass all the tests.

`ISAAC` was designed for cryptography by Jenkins [1996]. `AES` uses the Advanced Encryption standard based on the Rijndael cipher [NIST 2001; Daemen and Rijmen 2002] with a 128-bit key, in output feedback mode (OFB), in counter mode (CTR), and in key counter mode (KTR). The KTR mode uses a counter as key that is increased by 1 at each encryption iteration. AES under these modes was tested empirically by Hellekalek and Wegenkittl [2003]. Our implementation is based on the optimized C code for the Rijndael cipher written by Rijmen et al. [2000]. `AES` in CTR mode had a slightly suspect p-value of 9.2×10^{-5} for an overlapping collision test in *BigCrush*, but further replications of this test gave no further suspect value. `SHA-1` is based on the Secure Hash Algorithm SHA-1 [NIST 2002; Barker and Kelsey 2006] in output feedback mode (OFB) and in counter mode (CTR). These generators pass all the tests.

8. SUMMARY AND CONCLUSION

We have described the *TestU01* package for testing random number generators and used it to test several widely used or recently proposed generators. A minority of these generators pass all the tests (in the sense that no p-value is outside the interval $[10^{-10},\ 1 - 10^{-10}]$). The default generators of many popular software programs (e.g., Excel, MATLAB, Mathematica, Java standard library, R, etc.) fail several

tests miserably. The survivors are the long-period MRGs with good structure, the multiplicative lagged-Fibonacci generators, some nonlinear generators designed for cryptology, and some combined generators with components from different families. SWB and additive lagged-Fibonacci generators also pass when using appropriate decimation, but the decimation slows them down significantly. We believe that combined generators with components from different families should be given better attention because theoretical guarantees about their uniformity can be proved [L'Ecuyer and Granger-Piché 2003], their period can easily be made very long, splitting their period into long disjoint substreams is easy to do if we can do it for the components and it is not hard to select the components with this in mind. We emphasize that the RNGs implementations in *TestU01* are only for testing purposes. For simulation, we recommend RNGs with multiple streams and substreams such as those described in L'Ecuyer et al. [2002] and L'Ecuyer and Buist [2005], for example.

ACKNOWLEDGMENTS

This work has been supported by NSERC-Canada grant No. ODGP0110050 and a Canada Research Chair to the first author, and a Grant from Alcatel Canada. Part of the work was done while the first author was enjoying hospitality of IRISA, in Rennes, France. We thank the three anonymous reviewers for their constructive comments that helped improve the paper.

REFERENCES

Agarwal, R. C., Enenkel, R. F., Gustavson, F. G., Kothari, A., and m. Zubair. 2002. Fast pseudorandom-number generators with modulus 2^k or $2^k - 1$ using fused multiply-add. *IBM Journal of Research and Development 46,* 1, 97–116.

Anderson, T. W. and Darling, D. A. 1952. Asymptotic theory of certain goodness of fit criteria based on stochastic processes. *Annals of Mathematical Statistics 23*, 193–212.

Barker, E. and Kelsey, J. 2006. Recommendation for random number generation using deterministic random bit generators. SP-800-90, U.S. DoC/National Institute of Standards and Technology. See `http://csrc.nist.gov/publications/nistpubs/`.

Berlekamp, E. R. 1984. *Algebraic coding theory.* Aegean Park Press, Laguna Hills, CA, USA.

Bickel, P. J. and Breiman, L. 1983. Sums of functions of nearest neighbor distances, moment bounds, limit theorems and a goodness of fit test. *The Annals of Probability 11,* 1, 185–214.

Brent, R. P. 2004. Note on Marsaglia's xorshift random number generators. *Journal of Statistical Software 11,* 5, 1–4. See `http://www.jstatsoft.org/v11/i05/brent.pdf`.

Brown, F. B. and Nagaya, Y. 2002. The MCNP5 random number generator. Tech. Rep. LA-UR-02-3782, Los Alamos National Laboratory.

Carter, G. D. 1989. Aspects of local linear complexity. Ph.D. thesis, University of London.

Couture, R. and L'Ecuyer, P. 1994. On the lattice structure of certain linear congruential sequences related to AWC/SWB generators. *Mathematics of Computation 62,* 206, 798–808.

Couture, R. and L'Ecuyer, P. 1997. Distribution properties of multiply-with-carry random number generators. *Mathematics of Computation 66,* 218, 591–607.

Coveyou, R. R. 1969. Random number generation is too important to be left to chance. In *Applied Probability and Monte Carlo Methods and modern aspects of dynamics.* Studies in applied mathematics 3. Society for Industrial and Applied Mathematics, Philadelphia, 70–111.

Daemen, J. and Rijmen, V. 2002. *The Design of Rijndael.* Springer Verlag, New York. See `http://www.esat.kuleuven.ac.be/~rijmen/rijndael/`.

Darling, D. A. 1960. On the theorems of Kolmogorov-Smirnov. *Theory of Probability and Its Applications V,* 4, 356–360.

Deng, L.-Y. 2005. Efficient and portable multiple recursive generators of large order. *ACM Transactions on Modeling and Computer Simulation 15,* 1, 1–13.

Deng, L.-Y. and Lin, D. K. J. 2000. Random number generation for the new century. *The American Statistician 54,* 2, 145–150.

Durbin, J. 1973. *Distribution Theory for Tests Based on the Sample Distribution Function.* SIAM CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM, Philadelphia.

Eichenauer, J. and Lehn, J. 1986. A nonlinear congruential pseudorandom number generator. *Statistische Hefte 27*, 315–326.

Eichenauer-Herrmann, J. 1992. Inversive congruential pseudorandom numbers: A tutorial. *International Statistical Reviews 60*, 167–176.

Erdmann, E. D. 1992. Empirical tests of binary keystreams. M.S. thesis, Department of Mathematics, Royal Holloway and Bedford New College, University of London.

Feller, W. 1968. *An Introduction to Probability Theory, Vol. 1*, third ed. Wiley, New York.

Ferrenberg, A. M., Landau, D. P., and Wong, Y. J. 1992. Monte Carlo simulations: Hidden errors from "good" random number generators. *Physical Review Letters 69,* 23, 3382–3384.

Fishman, G. S. 1990. Multiplicative congruential random number generators with modulus 2^β: An exhaustive analysis for $\beta = 32$ and a partial analysis for $\beta = 48$. *Mathematics of Computation 54,* 189 (Jan), 331–344.

Fishman, G. S. 1996. *Monte Carlo: Concepts, Algorithms, and Applications.* Springer Series in Operations Research. Springer-Verlag, New York.

Fishman, G. S. and Moore III, L. S. 1986. An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. *SIAM Journal on Scientific and Statistical Computing 7,* 1, 24–45.

Földes, A. 1979. The limit distribution of the length of the longest head run. *Period Math. Hung. 10,* 4, 301–310.

Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Booth, M., and Rossi, F. 2004. GSL – GNU Scientific Library: Reference manual. Downloadable from `http://www.gnu.org/software/gsl/`.

Gammel, B. M. 2005. Matpack C++ Numerics and Graphics Library, Release 1.8.1. Downloadable from `http://users.physik.tu-muenchen.de/gammel/matpack/index.html`.

Goldreich, O. 1999. *Modern Cryptography, Probabilistic Proofs and Pseudo-randomness.* Springer Verlag, Berlin.

Good, I. J. 1953. The serial test for sampling numbers and other tests for randomness. *Proceedings of the Cambridge Philos. Society 49*, 276–284.

Gordon, L., Schilling, M. F., and Waterman, S. 1986. An extreme value theory for long head runs. *Probability Theory and Related Fields 72*, 279–287.

Greenwood, R. E. 1955. Coupon collector's test for random digits. *Math. Tables and other Aids to Computation 9*, 1–5, 224, 229.

Hellekalek, P. 1995. Inversive pseudorandom number generators: Concepts, results, and links. In *Proceedings of the 1995 Winter Simulation Conference*, C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman, Eds. IEEE Press, 255–262.

Hellekalek, P. 1998. Don't trust parallel Monte Carlo! In *Twelfth Workshop on Parallel and Distributed Simulation, May 1998, Banff, Canada.* IEEE Computer Society, Los Alamitos, California, 82–89.

Hellekalek, P. and Wegenkittl, S. 2003. Empirical evidence concerning AES. *ACM Transactions on Modeling and Computer Simulation 13,* 4, 322–333.

Holian, B. L., Percus, O. E., Warnock, T. T., and Whitlock, P. A. 1994. Pseudorandom number generator for massively parallel molecular-dynamics simulations. *Physical Review E 50,* 2, 1607–1615.

IBM 1968. *System/360 Scientific Subroutine Package, Version III, Programmer's Manual.* IBM, White Plains, New York.

IMSL. 1997. *IMSL STAT/LIBRARY.* Visual Numerics Inc., Houston, Texas, USA. `http://www.vni.com/books/dod/pdf/STATVol_2.pdf`.

Intel. 2003. Vector Statistical Library notes. Tech. Rep. Version 3, Intel Corporation. `http://www.intel.com/software/products/mkl/docs/vslnotes.pdf`.

James, F. 1994. RANLUX: A Fortran implementation of the high-quality pseudorandom number generator of Lüscher. *Computer Physics Communications 79*, 111–114.

Jenkins, B. 1996. ISAAC. In *Fast Software Encryption, Proceedings of the Third International Workshop, Cambridge, UK*, D. Gollmann, Ed. Lecture Notes in Computer Science, vol. 1039. Springer-Verlag, 41–49. Available at `http://burtleburtle.net/bob/rand/isaacafa.html`.

Kendall, M. G. and Babington-Smith, B. 1939. Second paper on random sampling numbers. *Journal of the Royal Statistical Society Supplement 6*, 51–61.

Kirkpatrick, S. and Stoll, E. 1981. A very fast shift-register sequence random number generator. *Journal of Computational Physics 40*, 517–526.

Kirschenhofer, P., Prodinger, H., and Szpankowski, W. 1994. Digital search trees again revisited: The internal path length perspective. *SIAM Journal on Computing 23*, 598–616.

Knuth, D. E. 1981. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Second ed. Addison-Wesley, Reading, Mass.

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

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

L'Ecuyer, P. 1988. Efficient and portable combined random number generators. *Communications of the ACM 31,* 6, 742–749 and 774. See also the correspondence in the same journal, 32, 8 (1989) 1019–1024.

L'Ecuyer, P. 1990. Random numbers for simulation. *Communications of the ACM 33,* 10, 85–97.

L'Ecuyer, P. 1992. Testing random number generators. In *Proceedings of the 1992 Winter Simulation Conference*. IEEE Press, 305–313.

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

L'Ecuyer, P. 1996a. Combined multiple recursive random number generators. *Operations Research 44,* 5, 816–822.

L'Ecuyer, P. 1996b. Maximally equidistributed combined Tausworthe generators. *Mathematics of Computation 65,* 213, 203–213.

L'Ecuyer, P. 1997a. Bad lattice structures for vectors of non-successive values produced by some linear recurrences. *INFORMS Journal on Computing 9,* 1, 57–60.

L'Ecuyer, P. 1997b. Tests based on sum-functions of spacings for uniform random numbers. *Journal of Statistical Computation and Simulation 59*, 251–269.

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. 2001. Software for uniform random number generation: Distinguishing the good and the bad. In *Proceedings of the 2001 Winter Simulation Conference*. IEEE Press, Pistacaway, NJ, 95–105.

L'Ecuyer, P. 2004. Random number generation. In *Handbook of Computational Statistics*, J. E. Gentle, W. Haerdle, and Y. Mori, Eds. Springer-Verlag, Berlin, 35–70. Chapter II.2.

L'Ecuyer, P. circa 2006. Uniform random number generation. In *Stochastic Simulation*, S. G. Henderson and B. L. Nelson, Eds. Handbooks of Operations Research and Management Science. Elsevier Science. chapter 3, to appear.

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

L'Ecuyer, P., Blouin, F., and Couture, R. 1993. A search for good multiple recursive random number generators. *ACM Transactions on Modeling and Computer Simulation 3,* 2, 87–98.

L'Ecuyer, P. and Buist, E. 2005. Simulation in Java with SSJ. In *Proceedings of the 2005 Winter Simulation Conference*. IEEE Press, 611–620.

L'Ecuyer, P., Cordeau, J.-F., and Simard, R. 2000. Close-point spatial tests and their application to random number generators. *Operations Research 48,* 2, 308–317.

L'Ecuyer, P. and Granger-Piché, J. 2003. Combined generators with components from different families. *Mathematics and Computers in Simulation 62*, 395–404.

L'Ecuyer, P. and Hellekalek, P. 1998. Random number generators: Selection criteria and testing. In *Random and Quasi-Random Point Sets*, P. Hellekalek and G. Larcher, Eds. Lecture Notes in Statistics, vol. 138. Springer-Verlag, New York, 223–265.

L'Ecuyer, P. and Simard, R. 1999. Beware of linear congruential generators with multipliers of the form $a = \pm 2^q \pm 2^r$. *ACM Transactions on Mathematical Software 25,* 3, 367–374.

L'Ecuyer, P. and Simard, R. 2001. On the performance of birthday spacings tests for certain families of random number generators. *Mathematics and Computers in Simulation 55,* 1–3, 131–137.

L'Ecuyer, P., Simard, R., Chen, E. J., and Kelton, W. D. 2002. An object-oriented random-number package with many long streams and substreams. *Operations Research 50,* 6, 1073–1075.

L'Ecuyer, P., Simard, R., and Wegenkittl, S. 2002. Sparse serial tests of uniformity for random number generators. *SIAM Journal on Scientific Computing 24,* 2, 652–668.

L'Ecuyer, P. and Touzin, R. 2000. Fast combined multiple recursive generators with multipliers of the form $a = \pm 2^q \pm 2^r$. In *Proceedings of the 2000 Winter Simulation Conference*, J. A. Joines, R. R. Barton, K. Kang, and P. A. Fishwick, Eds. IEEE Press, Pistacaway, NJ, 683–689.

L'Ecuyer, P. and Touzin, R. 2004. On the Deng-Lin random number generators and related methods. *Statistics and Computing 14*, 5–9.

Lewis, P. A. W., Goodman, A. S., and Miller, J. M. 1969. A pseudo-random number generator for the system/360. *IBM System's Journal 8*, 136–143.

Liang, Y. and Whitlock, P. A. 2001. A new empirical test for parallel pseudo-random number generators. *Mathematics and Computers in Simulation 55,* 1–3, 149–158.

Lüscher, M. 1994. A portable high-quality random number generator for lattice field theory simulations. *Computer Physics Communications 79*, 100–110.

Maplesoft. 2006. *Maple User Manual*. Waterloo Maple Inc., Waterloo, Canada. See `http://www.maplesoft.com/products/maple/`.

Marsaglia, G. 1972. The structure of linear congruential sequences. In *Applications of Number Theory to Numerical Analysis*, S. K. Zaremba, Ed. Academic Press, London, 249–285.

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

Marsaglia, G. 1996. DIEHARD: a battery of tests of randomness. See `http://stat.fsu.edu/~geo/diehard.html`.

Marsaglia, G. 1997. A random number generator for C. Posted to the electronic billboard `sci.math.num-analysis`.

Marsaglia, G. 1999. Random numbers for C: The END? Posted to the electronic billboard `sci.crypt.random-numbers`.

Marsaglia, G. 2002. Good 64-bit RNG's. Posted to the electronic billboard `sci.crypt.random-numbers`.

Marsaglia, G. 2003. Xorshift RNGs. *Journal of Statistical Software 8,* 14, 1–6. See `http://www.jstatsoft.org/v08/i14/xorshift.pdf`.

Marsaglia, G., Ananthanarayanan, K., and Paul, N. 1973. How to use the McGill Random Number Package "SUPER-DUPER". Tech. rep., School of Computer Science, McGill University, Montreal, Canada.

Marsaglia, G., Narasimhan, B., and Zaman, A. 1990. A random number generator for PC's. *Computer Physics Communications 60*, 345–349.

Marsaglia, G. and Tsang, W. W. 2002. Some difficult-to-pass tests of randomness. *Journal of Statistical Software 7,* 3, 1–9. See `http://www.jstatsoft.org/v07/i03/tuftests.pdf`.

Marsaglia, G. and Tsay, L.-H. 1985. Matrices and the structure of random number sequences. *Linear Algebra and its Applications 67*, 147–156.

Marsaglia, G. and Zaman, A. 1991. A new class of random number generators. *The Annals of Applied Probability 1*, 462–480.

Marsaglia, G. and Zaman, A. 1993a. The KISS generator. Tech. rep., Department of Statistics, University of Florida.

Marsaglia, G. and Zaman, A. 1993b. Monkey tests for random number generators. *Computers Math. Applic. 26,* 9, 1–10.

Mascagni, M. and Srinivasan, A. 2000. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software 26*, 436–461.

Massey, J. L. 1969. Shift-register synthesis and BCH decoding. *IEEE Trans. Inf. Theor IT-15*, 122–127.

MathSoft Inc. 2000. *S-PLUS 6.0 Guide to Statistics*. Vol. 2. Data Analysis Division, Seattle, WA.

Matsumoto, M. and Kurita, Y. 1992. Twisted GFSR generators. *ACM Transactions on Modeling and Computer Simulation 2,* 3, 179–194.

Matsumoto, M. and Kurita, Y. 1994. Twisted GFSR generators II. *ACM Transactions on Modeling and Computer Simulation 4,* 3, 254–266.

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

Maurer, J., Abrahams, D., Dawes, B., and Rivera, R. 2004. Boost random number library. Downloadable from `http://www.boost.org/libs/random/index.html`.

Maurer, U. M. 1992. A universal statistical test for random bit generators. *Journal of Cryptology 5,* 2, 89–105.

Moler, C. 2004. *Numerical Computing with MATLAB*. SIAM, Philadelphia.

NAG. 2002. *The NAG C Library Manual, Mark 7*. The Numerical Algorithms Group. `http://www.nag.co.uk/numeric/cl/manual/pdf/G05/g05cac.pdf` and `http://www.nag.co.uk/numeric/fl/manual/pdf/G05/g05kaf.pdf`.

Niederreiter, H. 1991. The linear complexity profile and the jump complexity of keystream sequences. In *Advances in Cryptology: Proceedings of EUROCRYPT'90*. Springer-Verlag, Berlin, 174–188.

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

NIST. 2001. Advanced encryption standard (AES). FIPS-197, U.S. DoC/National Institute of Standards and Technology. See `http://csrc.nist.gov/CryptoToolkit/tkencryption.html`.

NIST. 2002. Secure hash standard (SHS). FIPS-186-2, with change notice added in february 2004, U.S. DoC/National Institute of Standards and Technology. See `http://csrc.nist.gov/CryptoToolkit/tkhash.html`.

Panneton, F. and L'Ecuyer, P. 2005. On the xorshift random number generators. *ACM Transactions on Modeling and Computer Simulation 15,* 4, 346–361.

Panneton, F., L'Ecuyer, P., and Matsumoto, M. 2006. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software 32,* 1, 1–16.

Percus, O. E. and Whitlock, P. A. 1995. Theory and application of Marsaglia's monkey test for pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation 5,* 2, 87–100.

Press, W. H. and Teukolsky, S. A. 1992. *Numerical Recipes in C*. Cambridge University Press, Cambridge.

Project, T. G. 2003. *R: An Environment for Statistical Computing and Graphics*. The Free Software Foundation. Version 1.6.2. See `http://www.gnu.org/directory/GNU/R.html`.

Read, T. R. C. and Cressie, N. A. C. 1988. *Goodness-of-Fit Statistics for Discrete Multivariate Data*. Springer Series in Statistics. Springer-Verlag, New York.

Rijmen, V., Bosselærs, A., and Barreto, P. 2000. Optimised ANSI C code for the Rijndael cipher (now AES). Public domain software.

Ripley, B. D. 1990. Thoughts on pseudorandom number generators. *Journal of Computational and Applied Mathematics 31*, 153–163.

Ripley, B. D. and Venables, W. N. 1994. *Modern applied statistics with S-Plus.* Springer-Verlag, New York.

Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., and Vo, S. 2001. A statistical test suite for random and pseudorandom number generators for cryptographic applications. NIST special publication 800-22, National Institute of Standards and Technology (NIST), Gaithersburg, Maryland, USA. See `http://csrc.nist.gov/rng/`.

Rukhin, A. L. 2001. Testing randomness: A suite of statistical procedures. *Theory of Probability and Its Applications 45,* 1, 111–132.

Ryabko, B. Y., Monarev, V. A., and Shokin, Y. I. 2005. A new type of attack on block ciphers. *Problems of Information Transmission 41,* 4, 385–394.

Ryabko, B. Y., Stognienko, V. S., and Shokin, Y. I. 2004. A new test for randomness and its application to some cryptographic problems. *Journal of statistical planning and inference 123*, 365–376.

SciFace Software. 2004. *MuPAD.* SciFace Software GmbH & Co.KG, Paderborn, Germany. See `http://www.mupad.de/home.html`.

Sinclair, C. D. and Spurr, B. D. 1988. Approximations to the distribution function of the Anderson-Darling test statistic. *Journal of the American Statistical Association 83,* 404, 1190–1191.

Stephens, M. A. 1970. Use of the Kolmogorov-Smirnov, Cramér-Von Mises and related statistics without extensive tables. *Journal of the Royal Statistical Society, Series B 33,* 1, 115–122.

Stephens, M. S. 1986a. Tests based on EDF statistics. In *Goodness-of-Fit Techniques*, R. B. D'Agostino and M. S. Stephens, Eds. Marcel Dekker, New York and Basel.

Stephens, M. S. 1986b. Tests for the uniform distribution. In *Goodness-of-Fit Techniques*, R. B. D'Agostino and M. S. Stephens, Eds. Marcel Dekker, New York and Basel, 331–366.

Takashima, K. 1996. Last visit time tests for pseudorandom numbers. *J. Japanese Soc. Comp. Satist. 9,* 1, 1–14.

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

Tezuka, S., L'Ecuyer, P., and Couture, R. 1994. On the add-with-carry and subtract-with-borrow random number generators. *ACM Transactions of Modeling and Computer Simulation 3,* 4, 315–331.

Tootill, J. P. R., Robinson, W. D., and Eagle, D. J. 1973. An asymptotically random Tausworthe sequence. *Journal of the ACM 20*, 469–481.

Ugrin-Sparac, G. 1991. Stochastic investigations of pseudo-random number generators. *Computing 46*, 53–65.

Vattulainen, I., Ala-Nissila, T., and Kankaala, K. 1995. Physical models as tests of randomness. *Physical Review E 52,* 3, 3205–3213.

Wang, M. Z. 1997. Linear complexity profiles and jump complexity. *Information Processing Letters 61*, 165–168.

Wegenkittl, S. 1998. Generalized ϕ-divergence and frequency analysis in Markov chains. Ph.D. thesis, University of Salzburg. `http://random.mat.sbg.ac.at/team/`.

Wegenkittl, S. 2001. Entropy estimators and serial tests for ergodic chains. *IEEE Transactions on Information Theory 47,* 6, 2480–2489.

Wegenkittl, S. 2002. A generalized ϕ-divergence for asymptotically multivariate normal models. *Journal of Multivariate Analysis 83*, 288–302.

Wichmann, B. A. and Hill, I. D. 1982. An efficient and portable pseudo-random number generator. *Applied Statistics 31*, 188–190. See also corrections and remarks in the same journal by Wichmann and Hill, **33** (1984) 123; McLeod **34** (1985) 198–200; Zeisel **35** (1986) 89.

Wu, P.-C. 1997. Multiplicative, congruential random number generators with multiplier $\pm 2^{k_1} \pm 2^{k_2}$ and modulus $2^p - 1$. *ACM Transactions on Mathematical Software 23,* 2, 255–265.

Ziff, R. M. 1998. Four-tap shift-register-sequence random-number generators. *Computers in Physics 12,* 4, 385–392.

Ziv, J. and Lempel, A. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. on Information Theory 24,* 5, 530–536.