Computers Math. Applic. Vol. 26, No. 9, pp. 1–10, 1993
Printed in Great Britain. All rights reserved

0898-1221/93 $6.00 + 0.00

Monkey Tests for Random Number Generators

G. MARSAGLIA AND A. ZAMAN
Supercomputer Computations Research Institute
and
Department of Statistics, The Florida State University
Tallahasse, FL 32306, U.S.A.

(Received and accepted May 1993)

Abstract—This article describes some very simple, as well as some quite sophisticated, tests that shed light on the suitability of certain random number generators.

INTRODUCTION

Few images invoke the mysteries and ultimate certainties of a sequence of random events as well as that of the proverbial monkey at a typewriter. Surprisingly, many questions about the monkey's literary output—the times between appearances of certain strings, the number of distinct four-letter words in a million keystrokes, the time needed to spell CAT, for example—are well suited for assessing both uniformity and independence in the output of a random number generator (the monkey). Technically, we are concerned with overlapping m-tuples of successive elements in a random sequence.

For years, in our annual course Computer Methods in Probability and Statistics, we called these Overlapping m-Tuple Tests. But for the last few years we have used the monkey metaphor. It seems a better way to stimulate the interests of the students and, by invoking an interesting image, make them more readily accept the ideas and even feel as though they were their own.

We hope it will have a similar effect on you, the reader—not that we necessarily equate you with the students (or the monkey).

This article describes some very simple, as well as some quite sophisticated, tests that shed light on the suitability of certain random number generators. The generators are used to provide the random keystrokes for our monkey. The keyboards range from the standard 26 upper-case letters to an organ-like keyboard with 1024 keys to the DNA keyboard with four keys: C,G,A,T.

CAT TESTS

Now, to business. We start with an idea that provides a very inefficient test, but one that some random number generators (RNG's) fail. Our monkey (RNG) has a typewriter with 26 upper-case letters A,B,...,Z that he strikes at random. (Assume our RNG monkey produces uniform reals in [0,1), say by means of a procedure UNI(). The integer part of 26.*UNI() provides the random keystroke.) Now the CAT test: how many keys must the monkey strike until he spells CAT?

There are $26^3 = 17,576$ possible 3-letter words, so the average number of keystrokes necessary to produce CAT should be around 17,576. We will provide exact and approximate distributions below, and more efficient tests; for now, let's try this simple CAT test for a few common RNG's.

Research supported by NSF Grant DMS-9206972.

Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX

The congruential monkey, $I = 69069I \bmod 2^{32}$, converting to a real UNI on [0,1), gets CAT after 13,561 keystrokes, then after 18,263, then another 14,872 strokes produces the third CAT. Quite satisfactory.

Now consider the shift-register (Tauseworthe) monkey that produces 31-bit integers by exclusive or's, left shift 28 and right shift 3. This is the very generator suggested [1] as a replacement for congruential generators after discovery of their lattice structure [2]. The shift-register monkey never spells CAT, even after two million keystrokes. He can't get KAT, DOG, GOD, SEX, WOW or ZIG either. But he can get ZAG, and too often—every few thousand keystrokes. Indeed, it turns out that this monkey was only able to get 7,834 of the possible 17,576 3-letter words, (in a string of 1,000,000 keystrokes) and of course with his limited vocabulary, he gets those words too often.

Note that inability to get CAT should not be attributed to the equivalent of broken keys on the typewriter. This monkey still types each of the letters A to Z with the expected frequencies (and thus would pass a standard test for letter frequency). For example, 26,000 keystrokes produced 984 C's, 967 A's and 1021 T's, quite satisfactory. Yet continuing the run to 2,600,000 keystrokes failed to produce a single CAT!

As silly as it seems, this is a very effective and convincing way to show the unsuitability of certain random number generators. You may easily write a program and try it yourself. Although details have been lost, we remember using the CAT test to shoot down RNG's provided with Apple and Radio Shack computers (TRS80's ?) when they first came out in the 1970's.

MORE EFFICIENT TESTS

Most of our RNG monkey's output is wasted if we only count the number of keystrokes between successive appearances of CAT, DOG or some such 3-letter word. So, instead, suppose we count the frequencies of all the possible 3-letter words produced in a string of, say 1,757,603 keystrokes (the extra 3 so that we have exactly 1,757,600 3-letter words, and our expected frequency is 100 for each particular word). We need an array of 17576 elements to count the frequencies. Let $x_1, x_2, \ldots, x_{17576}$ be the observed counts.

Technically, the x's are asymptotically jointly normally distributed (they are m-dependent variables), and the appropriate statistical test is the quadratic form in the weak inverse of the covariance matrix of the x's, that is the quadratic form in the variables x_1–100, x_2–100, $\ldots$, x_{17576}–100 with coefficients the elements of any weak inverse C^- of the covariance matrix of the x's. (If that covariance matrix is C, then a weak inverse C^- satisfies $CC^-C = C$, and the value of the quadratic form is invariant (with probability 1) under choice of C^-.)

Note that the quadratic form Σ^2(observed-expected)/expected, Pearson's classical chi-square, is not the appropriate test. That test applies only for the classical occupancy problem (multinomial distribution), wherein balls are put into cells independently. Here we do not have independence; incrementing the count for, say, QCA certainly influences a possible increment for CAA,CAB,...,CAZ, whichever the next keystroke brings, but does not influence possible increments to the other cells.

But it turns out that we get the correct (or rather, a correct) quadratic form by applying Pearson's form twice: If we naively get Pearsons quadratic form for the 3-letter words, say

$$Q_3 = \sum_{i=1}^{i=17576} \frac{(x_i - 100)^2}{100},$$

and if we also count the frequencies of 2-letter words in the first 1,757,602 keystrokes and use the naive Pearson form, say

$$Q_2 = \sum_{i=1}^{i=676} \frac{(y_i - 1000)^2}{1000},$$

where $y_1, y_2, \ldots, y_{676}$ are the counts for the possible 2-letter words, then the difference, $Q_3 - Q_2$, is a zero-centered quadratic form in a weak inverse of the covariance matrix of the counts $x_1, \ldots, x_{17576}$, and is the appropriate (likelihood ratio) test that the x's came from a normal distribution with the specified means and covariance matrix.

If the hypothesis is true (the monkey is striking the keys uniformly *and independently*), then $Q_3 - Q_2$ will have a chi-square distribution with $26^3 - 26^2$ degrees of freedom, (the rank of C).

SPARSE OCCUPANCY TESTS: OPSO, OTSO, OQSO AND DNA

What if we want to apply the above test to, say, 4-letter words? We might take a string of two million keystrokes and count the frequencies of all the possible $26^4 = 456976$ 4-letter words. But that requires an array of 456,976 elements. That many bytes will do, if the frequencies are 100 or so. That's half a meg of bytes—quite a few. We might manage with a carefully tailored program, but we are pressing the limit. Counting 5-tuple or 6-tuple frequencies seems beyond reach. (Note that, in order to invoke asymptotic joint normality for the m-dependent cell counts, we need expected counts of some 20 or more, so we will need at least 6 bits for each cell.)

And yet 4-,5-,6-tuples, and longer, are where we are more likely to find unacceptable output from our RNG monkeys. What to do? Our approach is this: instead of counting frequencies of, say, 4-letter words in a long string of keystrokes, requiring a memory location—or at least a byte—for each possible 4-letter word, why not just count the presence or absence of each possible word? That requires a single bit for each possible word, or $26^4 = 456796$ bits for 4-letter words in an alphabet of 26 letters. That's about 14,000 computer (32-bit) words, a reasonably-sized array for most high level languages.

We call these *sparse-occupancy* tests. Because counting actual frequencies requires arrays too large, we only count the number of empty cells, that is, in a long string of keystrokes, we use a bit map to find how many 4- (5-,6- or higher-) letter words are missing. Some interesting probability theory is required to develop appropriate tests.

THE OPSO TEST

Here OPSO means Overlapping-Pairs-Sparse-Occupancy. We observe 2-letter words in a long sequence of keystrokes from an alphabet of 2^{10} letters. By setting a bit, we mark the presence or absence of every possible 2-letter word. The first 10 bits (or any particular 10 bits) of an integer produced by a generator (monkey) determines the keystroke. (We abandon the 26-letter alphabet from here on; alphabets of size $2, 2^2, 2^3, \ldots, 2^{10}$ are better suited for testing RNG's.)

Before developing the theory, here is an example of the OPSO test: We generate 2^{21} keystrokes from an alphabet of $2^{10} = 1024$ letters. Each keystroke is produced by the first 10 bits of the congruential generator $x_n = 69069x_{n-1} \bmod 2^{32}$. We count the number of missing 2-letter words. As we shall see, the number of missing 2-letter words should be approximately normal with $\mu = 141909$ and $\sigma = 290$. Our first run (with seed value 1234567) of 2^{21} keystrokes has 141,979 missing 2-letter words, corresponding to a standard normal variate of .240, the next three runs of 2^{21} keystrokes have 141980, 141753 and 141785 missing 2-letter words, corresponding to standard normal values .243, −.535, −.428. Good. Our 69069 congruential monkey comes through again.

Now let's put the 31-bit left-28-right-3 shift-register monkey at the keyboard. We already saw he couldn't manage CAT on a 26-key typewriter; how will he do on a veritable organ with 1024 keys? Lousy! With a seed value of 1234567, his first string of 2^{21} keystrokes had 1,032,192 missing 2-letter words, some 3070 sigmas from the mean. The second was as bad.

Next, let's import a monkey from Berkeley, with keystrokes determined by the leading 10 bits from the Berkeley Unix RNG $x_n = 62605x_{n-1} + 113218009 \bmod 2^{29}$. Again with seed 1234567, six runs of 2^{21} keystrokes produced what should be the equivalent of six independent standard normal values: −1.771, −3.447, −1.585, −2.903, −1.757, −2.370. Not good; this Berkeley monkey fails the OPSO test, but it is not the spectacular failure of the shift-register monkey.

Another shift-register monkey: left shift 18, right shift 13 on 31 bit words. This monkey uses what we have found to be the best of the shift-register generators. How will she do on OPSO? With seed value 1234567, our monkey provides 2^{21} keystrokes (from leading 10 bits) that have 139,375 missing 2-letter words. That is -8.74 sigmas from the mean. Bad. The next run of 2^{21} keystrokes has 139,946 missing 2-letter words: -6.77 sigmas of 290 from the mean of 141,909. Even the best of the shift-register generators fails the OPSO test, but not in as spectacular fashion as those for other shift register generators.

OPSO THEORY

The OPSO test counts the number of missing 2-letter words in a long string of n random keystrokes. If $n = 2^{21} = 2,097,152$ and there are $\alpha = 2^{10} = 1024$ letters in the alphabet, then the number of missing words should average 141,909 with a standard deviation of 290. How is this determined?

The answer: not easily. At least, the variance is not easy; the mean *is* easy. To get the mean, we take advantage of the near lack-of-memory property of the monkey's output. If he has not typed a particular word after, say, 1000 keystrokes, then the distribution of the time remaining until he does has virtually the same distribution as the original. In other words, the time until the monkey types a particular 2-letter word should be close to exponential, with mean $\mu = \alpha^2 = 2^{20}$, and the probability he does not type the word within n keystrokes should be $e^{-n/\mu}$, to considerable accuracy.

To determine that accuracy, we need the true probability that n keystrokes will not produce a particular 2-letter word. There are two kinds of 2-letter words: AB and AA. The probability of no AB in n keystrokes is

$$\text{The coefficient of } z^n \text{ in the Taylor expansion of } \frac{1}{1 - z + p^2 z^2},$$

where $p = 2^{-10}$, the probability for each of the keystrokes.

The general form of the required probability is $c_1 r_1^n + c_2 r_2^n$, with r_1, r_2 the roots of $x^2 - x + p^2 = 0$ and $c_i = r_i/(r_i - p^2)$. Even with n as small as 100 this becomes, to great accuracy, $c_1 r_1^n$, with r_1 the dominant root:

$$\Pr(\text{no AB in } n \text{ strokes}) = 1.000001907354(.99999904632477409731349945987)^n.$$

When $n = 2^{21}$ this becomes .135335154, while $e^{-2} = .135335283$.

The probability of no AA in n keystrokes is slightly different,

$$\text{The coefficient of } z^n \text{ in the Taylor expansion of } \frac{1 + pz}{1 - z - (1 - p)z - (p - p^2)z^2}.$$

For large n, this becomes

$$\Pr(\text{no AA in } n \text{ strokes}) = 1.0000019045637(.99999904725519072388710142577)^n.$$

When $n = 2^{21}$ this becomes .135599426952, compared to $e^{-2} = .135335283$ that comes from the lack-of-memory assumption.

Now, to find the expected number of missing 2-letter words in a string of n keystrokes, we use indicator variables. Let $w_1, w_2, \ldots, w_{2097152}$ be the set of possible 2-letter words and let

$$x_i = 1 \text{ if word } w_i \text{ is missing, else } x_i = 0.$$

Then the number of missing words is $x_1 + \cdots + x_{2097152}$, and the expected number of missing words is

$$\mathrm{E}(x_1) + \cdots + \mathrm{E}(x_{2097152}).$$

There are $2^{20} - 2^{10}$ 2-letter words of type AB, and 2^{10} of type AA. When $n = 2^{21}$, our expected number of missing 2-letter words is

$$(2^{20} - 2^{10}) \times .13533515417056227716252 + 2^{10} \times .13559948118968844042057 4,$$

and this reduces to 141909.4652904189697.

On the other hand, the approximation based on the lack-of-memory assumption yields

$$2^{20} e^{-2^{21}/2^{20}} = 141909.32995511439001.$$

Thus, the easy method for finding the average number of missing 2-letter words from n keystrokes, $2^{20} e^{-n/2^{20}}$, is quite suitable for practical applications of the OPSO test.

Those not acquainted with methods for developing generating functions and solving recurrence equations such as those above may wish to look at the marvelous treatment in the book [3] developed out of Donald Knuth's concrete mathematics course at Stanford, in particular, sections 7.1–7.3 and 8.4.

Now for the hard part: the variance of the number of missing 2-letter words. It is the sum of $\text{cov}(x_i, x_j)$ for i and j each ranging from 1 to 2^{10}. There are 2^{20} such covariances. They fall into some 17 different types, with a generating function for each type. Thus the expected value of $x_i x_j$, with x_i associated with a word such as AB, and x_j associated with CA, requires a different generating function than does that associated with AA,BA.

We omit details of extensive calculations of the covariances for each of the 17 types. If their frequencies are accounted for, the total yields the required variance. It is 84255.766087785. Thus, with an alphabet of 2^{10} letters, the number of missing 2-letter words in a string of 2^{21} keystrokes has mean 141,909 and standard deviation 290.27. It has a distribution close enough to normal that the statistic $(x - 141909)/290$ is the appropriate one for the OPSO test; it should behave as a standard normal variate. Here x is the number of 2-letter words that are missing from the string of 2^{21} keystrokes. A value of $(x - 141909)/290$ in absolute value greater than, say, 3 is cause for concern. A really good monkey (RNG) would cause concern only a few times in 1000 tests.

Now an aside with a call for help: We have not been able to find a suitable justification for an approximation for the variance of the number of missing k-letter words in a long string of n keystrokes. Perhaps some reader can. By just fooling around with simple formulas, we find this: the variance is approximately

$$\alpha^k e^{-\lambda}(1 - 3e^{-\lambda}),$$

with α the number of letters in the alphabet and $\lambda = n/\alpha^k$. For example, with $\alpha = 2^{10}, n = 2^{21}$ this approximation yields 84293, for 2-letter words, compared to the true variance of 84255. The corresponding square roots are 290.27 and 290.33. Not bad. With an alphabet of $\alpha = 2^{21}$ and $n = 2^{23}$, the true mean and standard deviation are 567,639 and 580.8, whereas the formula $\alpha^2 e^{-\lambda}(1 - 3e^{-\lambda})$ yields an approximate standard deviation of 580.7.

We have not been able to find the true variance for OTSO, overlapping triples of 3-letter words, but sampling suggests its variance is consistent with the value given by the approximation. Similarly for OQSO, overlapping 4-letter words. Both OTSO and OQSO are discussed in more detail in the next sections.

But we can find no heuristic reason why, for k-letter words, $\alpha^k e^{-\lambda}(1 - 3e^{-\lambda})$ should give the approximate variance, with $\lambda = n/\alpha^k$. But it does. Perhaps it is just a coincidence for the range of alphabet sizes and keystroke lengths of the size suited for monkey tests. It certainly is not suitable unless $3e^{-\lambda} < 1$, and for very long sequences, with $\lambda = n/\alpha^2 > 6$ or so, the number of missing k-letter words becomes nearly Poisson distributed, with variance equal to the easily found mean. So a general variance-approximation formula of the form $\alpha^k e^{-\lambda}(1 - ce^{-\lambda})$ may do, with c ranging from 3 to 0. $c = 3$ does nicely for the tests here.

THE OTSO TEST

OTSO means Overlapping-Triples-Sparse-Occupancy, meaning the number of missing 3-letter words in a long string of n keystrokes. Constraints on the size of our bit map make $\alpha = 2^6$ a reasonable size for the alphabet, calling for an array of 2^{13} 32-bit words to record the presence or absence of each possible word. The RNG monkey's keystroke will be determined by six bits from the random integer—most often the leading six, the most important, bits. But every set of six bits should provide suitably random and independent keystrokes, if we are to have a truly satisfactory RNG.

So with a keyboard of $\alpha = 64$ keys our RNG monkey produces, say, $n = 2^{21}$ keystrokes and we count the number of missing 3-letter words. Enumerating the possible kinds of 3-letter words: AAA, AAB, ABA, BAA, ABC, we find generating functions for the probability that each particular type will be missing in n keystrokes, take into account their frequencies, and find the true expected number of missing 3-letter words in a string of 2^{21} keystrokes. It is 87.9393.

The value that results from the lack-of-memory assumption is $\alpha^3 e^{-n/\alpha^3} = 2^{18}e^{-8} = 87.9395$. So, once again, the approximation is very accurate.

It is a formidable task to find the exact variance for the number of missing 3-letter words in 2^{21} keystrokes. There are hundreds of different word-type pairs, such as ABA,CAB for example, for which generating functions must be found. Then asymptotic forms must be found and combined with the proper frequencies to get the true variance. We have a half-hearted project underway to automate the process with a computer program, but have progressed only to the we-think-it-is-feasible stage.

The formula $\alpha^3 e^{-\lambda}(1 - 3e^{-\lambda})$ gives 87.85, with resulting $\sigma = 9.37$. This is consistent with simulation results. But note that when $\lambda = 2^{21}/2^{18} = 8$, the factor $(1-3e^{-\lambda})$ is of no consequence. The estimated variance reduces to the mean value $\alpha^3 e^{-\lambda}$, which is what we expect if the number of missing 3-letter words has a limiting Poisson distribution.

Note that the OTSO test does not require the true variance. One may test that a set of x values, the number of missing 3-letter words in a string of 2^{21} keystrokes, comes from a normal distribution with mean 87.85 with a t-test, using cumulative estimates of the standard deviation from the sample values. Larger samples of x's are necessary to establish a test failure. The test is a bit sharper when we know the variance.

Here is a practical example of the OTSO test. A random number generator proposed (and touted on internet) by Haas [4], generates 4-digit ordinary decimal integers. Suppose we assume an alphabet of $\alpha = 100$ characters and let the last two digits of Haas-generated numbers determine our monkey's keystrokes, then consider 3-letter words. If we let the monkey produce 2,000,003 keystrokes, and hence 2 million 3-letter words, we expect there should be around $\alpha^3 e^{-2000000/1000000} = 135,335$ missing 3-letter words.

The first three runs produced 147,440, then 147,922, then 147,691. These are all at least 43 sigmas from the mean, a spectacular failure that a more extensive t-test will confirm. (The above empirical approximation for the variance suggests $\sigma = 283.5$ for this OTSO test.)

THE OQSO TEST

The acronym OQSO means Overlapping-Quadruples-Sparse-Occupancy—the number of missing 4-letter words in a long string of keystrokes. We use an alphabet of $\alpha = 2^5$ letters and a string of $n = 2^{21}$ keystrokes. Thus selecting any five bits from the integer produced by the RNG provides the resulting keystroke for our RNG monkey.

By enumerating the possible kinds of 4-letter words, finding their generating functions and asymptotic forms for the coefficients of z^n, then combining them with the proper frequencies, the true expected number of missing 4-letter words in a string of $n = 2^{21}$ keystrokes may be found to better than 40 digits of accuracy. To the first 11 digits, it is $\mu = 141909.47365\ldots$. The approximation based on the lack-of-memory property yields 141909.33.

We don't know—and doubt that we ever will know—the true variance. There are just too many kinds of pairs of 4-letter words to undertake finding all the necessary generating functions.

The approximation $\alpha^k e^{-\lambda}(1 - 3e^{-\lambda})$ yields 84293, with a resulting σ of 290.33, the same values as those for OPSO, since OPSO has the same values for α^k and λ. Simulation results are consistent with $\sigma = 290$ for OQSO, and we earlier found the true σ for OPSO to be 290, to three places.

The display below gives details of the enumeration necessary for finding the true expected number of missing 4-letter words, in order to assess the accuracy of the approximation arising from the lack-of-memory assumption.

For each type of word, we need the frequency, the generating function and the probability that a string of $n = 2^{21}$ keystrokes will not contain that word. The notation A$'$ means not-A, X and Y designate any letters of the alphabet. There are $(\alpha^4 - \alpha^3)$ words of type AXYA$'$, $(\alpha^3 - \alpha)$ of type AXYA, with XY not AA, and α words of type AAAA. For each particular 4-letter word there is a generating function $N(z)$. The coefficient of z^n in the series expansion of the generating function $N(z)$ gives the probability that a string of n keystrokes will not contain that particular 4-letter word.

THE EXPECTED NUMBER OF MISSING WORDS FROM A STRING OF $n = 2^{21}$ 4-LETTER WORDS. THE ALPHABET HAS $\alpha = 32$ LETTERS.

Form of word: AXYA$'$, ($\alpha^4 - \alpha^3$ of these)
$N(z) = 1/(1 - z + p^4 z^4)$
Dominant root: $r = .99999904632295509663 \sim 1 - p^4 - 3p^8$
Coefficient: $c = 1.00000381472273166747 \sim 1 + 4p^4 + 28p^8$
Contribution to μ: $(\alpha^4 - \alpha^3)cr^n = 137474.27007432892416$

Form of word: AXYA, with XY not AA ($\alpha^3 - \alpha$ of these)
$N(z) = (1 + p^3z^3)/(1 - z + p^3z^3 + (p^4 - p^3)z^4)$
Dominant root: $r = .99999904635205828873 \sim 1 - p^4 + p^7$
Coefficient: $c = 1.00000381451901068279 \sim 1 + 4p^4 - 7p^7$
Contribution to μ: $(\alpha^3 - \alpha)cr^n = 4430.59356383447583$

Form of word: AAAA, (α of these)
$N(z) = (1 + pz + p^2z^2 + p^3z^3)/(1 - (1-p)z - (p - p^2)z^2 - (p^2 - p^3)z^3 - (p^3 - p^4)z^4)$
Dominant root: $r = .99999907612459180309 7 \sim 1 - p^4 + p^5$
Coefficient: $c = 1.00000366571613359007 7 \sim 1 + p^4 - 5p^5$
Contribution to μ: $\alpha cr^n = 4.61001685622600658813 07$

Total: $\mu = 141909.47365$
Lack-of-memory approximation: $\alpha^4 e^{-n/\alpha^4} = 141909.33$

THE DNA TEST

If we have an alphabet of 4 letters: C,G,A,T, then a succession of keystrokes from that alphabet will look like the mapping of a segment of DNA. Hence the name. Our monkey generates long strings and we look for the incidence of 10-letter words. Thus we have 2^{20} possible 10-letter words, and we need an array of 2^{15} 32-bit words for mapping the presence or absence of all possible 10-letter words.

As with OTSO and OQSO, we can find the exact expected number of missing 10-letter words from a string of, say, $n = 2^{21}$ keystrokes, but finding the exact variance looks hopeless.

To find the true expected number of missing 10-letter words, we must consider 22 kinds of 10-letter words (depending on how many leading segments of each word match equal-length trailing segments), find their generating functions, the probability the word will not appear in $n = 2^{21}$ keystrokes, then combine all those probabilities, with appropriate frequencies.

The result is 141910.5378411, the expected number of missing 10-letter words in a random DNA segment of 2^{21} C's,G's,A's and T's. The expected value from the lack-of-memory property is the same as that for OPSO and OQSO, with $\alpha^k = 2^{20}$ and $\lambda = 2$: 141909.

The estimate of the variance from our formula $\alpha^{10}e^{-\lambda}(1 - 3e^{-\lambda})$ again yields a σ of 290. As with OQSO, extensive simulations show that the true σ must be very close to 290.

So a reasonble implementation of the DNA test is this: Generate 2^{21} keystrokes from the alphabet {C,G,A,T} (using two bits from each random integer), and let x be the number of missing 10-letter words. Do this, say 4 times. The resulting values $(x_1 - 141911)/290, \ldots, (x_4 - 141911)/290)$ should look like a sample of 4 independent standard normal variables.

DIFFERENCES IN THE TESTS

How do the tests OPSO, OTSO, OQSO and DNA differ? All of them count the number of missing k-letter words in a long string of n keystrokes from an alphabet of α letters, and thus test both uniformity and independence in the output of a random number generator.

OPSO uses more of each random number, 10 bits, but space and time limitations only allow testing 2-letter words, that is, independence for pairs (2 dimensions). On the other hand, DNA can test 10-tuples (10 dimensional behaviour) but, again from time and space considerations, at a depth of only 2 bits per random number. Some RNG's may do well in one of the tests and not in the other. OTSO and OQSO are in between, using 6 bits from each number for behaviour in 3 dimensions or 5 bits from each number for 4 dimensions.

Of course a good RNG monkey should pass all these tests. We next give examples of good and not so good RNG monkeys.

RESULTS OF EXTENSIVE MONKEY TESTS

We have applied various monkey tests to many different RNG's. We find OPSO,OQSO and DNA to be the most effective. Simple tests such as the CAT test of Section 1 are easily programmed, but more complicated programs such as those for OPSO, OQSO or DNA are really not that complicated, and the reader may find it easier to program his own versions rather than get them from us and figure out how to use them.

Next, let us suggest various levels of stringency for tests of a random number generator. For most applications, the leading bits, say the the first 10–16 bits, of a RNG are the most important and should pass all tests. Thus, for monkey tests, keystrokes determined by the leading bits of the random number must be satisfactorily random and independent. Such RNG's will be satisfactory for most purposes.

There are an increasing number of applications, however, where adequate randomness in the most significant bits of a random integer or uniform variate on [0,1) is not enough. Monte Carlo simulations using samples of several millions are examples, and another is use of the trailing byte of a random integer to provide the dominant step in generating a discrete random variable. (Probably the fastest method for generating a discrete variate is to express each of its probabilities as a sum $p_i = k_i/256 + r_i$. Then the required variate is generated most of the time by fetching a value from a table with index a random byte determined by the trailing 8 bits of a random 32-bit integer.)

So, a really good random number generator must pass monkey tests with keystrokes determined by all possible substrings of bits of its random numbers. Most, but not all, of the standard RNG's have satisfactory leading bits, but it is quite difficult to fashion generators with satisfactory trailing bits.

Thus, for example, a really good RNG must pass OPSO with keystrokes determined by bits 1–10, 2–11, . . . ,23–32. Similarly, it must pass OQSO with bits 1–5,2–6, . . . ,28–32 and DNA with bits 1–2,2–3, . . . ,31–32.

A few generators are that good, others are not. Here is a summary of kinds of generators and the extent to which they pass monkey tests. More detailed descriptions of the classes of generators are in [5–8]

SUMMARY FOR VARIOUS CLASSES OF RNG'S

Congruential Generators

These produce sequences $x_n = ax_{n-1} + b \bmod m$. The modulus m is best taken to be 2^{32} for speed and convenience in modern CPU's, but a prime modulus produces better trailing bits. Those with prime modulus seem to do well on all substrings of bits for the OPSO, OQSO and DNA tests. They would be the congruential generators of choice if they were not so awkward to implement.

Congruential generators with modulus a power of 2 usually do well if strings of bits from the first 10 or 12 are used to determine a keystroke. But note, from above, that the Berkeley congruential RNG failed OPSO with keystrokes determined by the leading ten bits. The widely used generator $x_n = 69069x_{n-1} \bmod 2^{32}$ (the system generator for Vax's) seems to pass all tests determined by leading bits, but it fails badly on OPSO, OQSO and DNA if bits beyond the first 13 are used.

Shift Register Generators

Practical versions of these generators are perhaps best described by viewing a computer word as a binary vector β. Then the sequence is $\beta, \beta T, \beta T^2, \ldots$, with T a binary matrix. Such a T is chosen to make the period long and implementation reasonable; among the most common T's are those of the examples above: $T = (I + R^{15})(I + L^{17})$ for 32-bit binary vectors, and $T = (I + R^{28})(I + L^3)$ or $T = (I + L^{18})(I + R^{13})$ for 31-bit vectors. R and L are matrices that effect a right (left) shift of one position in the binary vector, and addition of binary vectors is the exclusive-or operation. Shift register generators do poorly on most monkey tests, with the exception of $T = (I + L^{18})(I + R^{13})$ or its transpose, which do well on most, but not all—see, for example, its poor performance on OPSO at the end of Section 4.

Lagged Fibonacci Generators

These generators use an initial set of elements $x_1, x_2, \ldots, x_r$ and two "lags" r and s with $r > s$. Successive elements are generated by the recursion, for $n > r$: $x_n = x_{n-r} \diamond x_{n-s}$, where $\diamond$ is some binary operation. The initial (seed) elements are computer words and the binary operation might be +,−,∗ or ⊕ (exclusive-or). For + or −, the x's might be integers mod 2^k or single- or double-precision reals mod 1. For ∗, odd integers mod 2^k. We designate such a generator loosely as F(r,s,◇), although each lagged-Fibonacci generator depends on details of the particular binary operation and the finite set of elements it operates on.

Generators using multiplication on odd integers, automatically modulo 2^{32}, are the best of the lagged Fibonacci generators, passing other tests as well as monkey tests. (The last bit is always 1, so of course they cannot pass OPSO, OQSO or DNA if the last bit contributes to the keystroke.) Overall, F(r,s,+) and F(r,s,-) and F(r,s,⊕) do very well on monkey tests for all strings of bits. But F(r,s,⊕) may fail for pairs (r,s) such as (31,13) or (17,5) because of their their inadequate period, 2^r, in contrast to the other lagged Fibonacci generators, which have periods about 2^{32+r}.

AWC and SWB Generators

The recently developed add-with-carry (AWC) and subtract-with-borrow (SWB) generators [8] have remarkably long periods and seem to pass OPSO, OQSO and DNA tests for strings of bits from all parts of the random integer. They are based on recursions such as $x_n = x_{n-r} + x_{n-s} + c \bmod b$, where the '$c$' is not a constant, but the carry bit: 1 or 0, depending on whether the previous addition did, or did not, exceed the modulus b. That describes an AWC generator. An SWB generator forms $x_n = x_{n-s} - x_{n-r} - c \bmod b$, with c either 1 or 0, depending on whether the previous subtraction was negative.

An example: the SWB generator $x_n = x_{n-24} - x_{n-37} - c \bmod 2^{32}$ has period about 2^{1178} and passes all the CAT, OPSO, OQSO and DNA tests put to it, for all substrings of its 32-bit integers. (However, it is not perfect; it fails the birthday-spacing test described in [5], as do other AWC and SWB generators and lagged-Fibonacci generators using +,– or $\oplus$.)

Combination Generators

There is theoretical and empirical support for the idea that combining two random number generators produces a generator with better randomness than either of its components [5]. For example, consider the simple congruential generator $x_n = 69069x_{n-1} \bmod 2^{32}$ applied to OPSO for bits 15–24, OQSO for bits 15–18 and DNA for bits 15–16. It fails all of them spectacularly, producing, respectively, 984,860 and 983,840 and 983,840 missing words when 141,909 are expected. But if we combine it (by addition) with the shift register generator $\beta = \beta(I+L^{17})(I+R^{15})$ on 32 bits (which fails spectacularly on all substrings of bits), we get these missing-word counts for OPSO,bits 15-24: 141909; OQSO, bits 15-18: 142116; and DNA, bits 15-16: 141,524. Very good, being 0,.762 and -.092 standard deviations from the mean.

This example is for the widely used combination generator Super-Duper, combining $I = 69069*I$ with $\beta = \beta(I+L^{17})(I+R^{15})$. But even that combination fails monkey tests on trailing bits. The KISS generator (Keep-It-Simple-Stupid) is an example of a combination generator that is about as simple as one can be and have very long period, about 2^{95}, and pass all tests on all strings of bits. It combines a congruential and two shift register generators, $I = 69069 * I + 23606797$, $\beta = \beta(I + R^{17})(I + L^{15})$ and $\beta = \beta(I + L^{18})(I + R^{13})$, the latter on 31-bit words.

AVAILABILITY

Descriptions and code for the KISS generator and for ULTRA, a generator that combines the above SWB generator with $I = 69069I$ and has period some 10^{366}, are available from the authors (geo@stat.fsu.edu). Also available: our DIEHARD battery of tests, recently completed for distribution in "user friendly" form. It includes the monkey tests described here as well as other stringent tests described in [5] and our versions of standard tests.

References

1. J.R.B. Whittlesey, On the multidimensional uniformity of pseudorandom number generators, *Comm. ACM* **12**, 247, (1969).
2. G. Marsaglia, Random numbers fall mainly in the planes, *Proc. Nat. Acad. Sci.* **61**, 25–28, (1968).
3. R.L. Graham, D.E. Knuth and O. Patashnik, *Concrete Mathematics*, Addison-Wesley, Reading, MA, (1989).
4. A. Haas, The multiple prime random number generator, *ACM Transactions on Mathematical Software* **13** (4), (1987).
5. G. Marsaglia, Keynote address: A current view of random number generators, In *Proceedings, Computer Science and Statistics: 16th Symposium on the Interface*, Elsevier, (1985).
6. G. Marsaglia and L.H. Tsay, Matrices and the structure of random number sequences, *Linear Algebra and its Applications* **67**, 147–156, (1985).
7. G. Marsaglia, The mathematics of random number generators, *Proceedings of Symposia on Applied Mathematics* **46**, 73–89, (1992).
8. G. Marsaglia and A. Zaman, A new class of random number generators, *Annals of Applied Probability* **1** (3), 462–480, (1991).