Some difficult-to-pass tests of randomness

George Marsaglia *
The Florida State University
and
Wai Wan Tsang *
The University of Hong Kong

Abstract

We describe three tests of randomness—tests that many random number generators fail. In particular, all congruential generators—even those based on a prime modulus—fail at least one of the tests, as do many simple generators, such as shift register and lagged Fibonacci. On the other hand, generators that pass the three tests seem to pass all the tests in the Diehard Battery of Tests.

Note that these tests concern the randomness of a generator's output as a sequence of independent, uniform 32-bit integers. For uses where the output is converted to uniform variates in [0,1), potential flaws of the output as integers will seldom cause problems after the conversion. Most generators seem to be adequate for producing a set of uniform reals in [0,1), but several important applications, notably in cryptography and number theory—for example, establishing probable primes, complexity of factoring algorithms, random partitions of large integers—may require satisfactory performance on the kinds of tests we describe here.

1 Introduction

Judging from journal articles and the far more topical content of newsgroup exchanges, there is increasing interest in the use of random number generators (RNG's) and their suitability for various kinds of computer simulations and for use in cryptography and computational number theory. Much of the interest in RNG's may be classified under two headings:
(1) Which are good RNG's and (2) How do you decide?

For question (1), we have suggested a number of answers, see [3,5,7], and this note is directed toward question (2), for which we suggest three tests of randomness that serve to help distinguish the good from the not-so-good.

In an effort to provide tests that were more stringent than the usual easy-to-pass tests first described in [2] and extended in Knuth's widely read book [1], Marsaglia described several more tests in [3], then added others to make up the Diehard Battery of Tests of Randomness [6]. These tests were included in a CDROM produced under a grant from the National Science Foundation. The CDROM provided a reliable source of random bits (600 megabytes) as well as the Diehard Battery of Tests of Randomness. Some 1000 free copies were distributed to interested researchers worldwide.

The stock of free copies was soon exhausted, but the CDROM is readily available on the internet, from sites at Florida State University and the University of Hong Kong, and it has been accessed hundreds of thousands of times.

The three tests we describe here may be viewed as a distillation of the Diehard battery of tests: reduce the volume and concentrate the essence. Based on our experience, if a random number generator passes the three tests described here, it is likely to pass the tests in Diehard. In addition, this 'distilled version' of Diehard is much easier to apply, because unlike Diehard, which requires a binary file of some 12 megabytes of random bits (and the concern of many queries on creating such files), the three tests here merely require the name of the C function that provides the 32-bit random integers to be tested.

*Research supported by Innovation and Technology Support Programme, Government of Hong Kong, Grant ITS/277/0

In order to save journal space, specific versions of the three tests are not included, but C versions are readily available, and will be included in new versions of the Diehard CDROM available via the internet.

2 The gcd Test

We begin with an example: Consider integers $u = 297$ and $v = 366$. Use Euclid's algorithm to determine the gcd of u and v:

```
366 = 1*297 + 69
297 = 4*69  + 21
 69 = 3*21  + 6
 21 = 3*6   + 3
  6 = 2*3   + 0
```

If u and v are random 32-bit positive integers, then three objects result from applying Euclid's algorithm to the pair u, v: (1) the number the iterations, k, needed to find the gcd (k=5 in the above example), (2) a variable-length sequence of partial quotients (1,4,3,3,2 in the example) and (3) the gcd—the final value of u (3 in the example).

Repeating this procedure for many random choices of pairs u, v will produce three lists: (1) a list of k's that are independent and identically distributed (iid), (2) a list of variable-length sequences of partial quotients with elements not iid, and (3) a list of resulting gcd's that are iid.

Successive steps of Euclid's algorithm provide the partial quotients q of the continued fraction expansion of a ratio u/v. Such q's also arise as the partial quotients of the continued fraction expansion of a random real in (0,1]. In the latter case, the q's are nearly iid with a distribution conjectured by Gauss: $\Pr[q < x] = \ln(1+x)/\ln(2)$. A thorough treatment of Euclid's algorithm, its history and relation to continued fractions, are in Knuth [1], with old and new results (pages 333-379, with extensive "Answers to Exercises" in pages 640-657).

The q's that arise from a random $u \in \{1, 2, \ldots, 2^{32}-1\}$ and fixed $v = n = 2^{32}$ seem adequately close to iid with Gauss's conjectured distribution. But the q's that arise from both u and v randomly chosen from 1 to $n = 2^{32}-1$, although they also seem to be close to iid, have a different distribution, the mixture with weights $1/n$ of the distributions with fixed v from 1 to n and u random from 1 to v. We have found that apparent distribution empirically. But we have not included a test on the distribution for such q's, as it did not turn out to be as discriminating as do tests on k and the gcd.

Thus we restrict ourselves to the true iid situations: the k values and the gcd values that result from many random choices of u and v in $\{1, 2, \ldots, 2^{32}-1\}$. It turns out that if we choose u and v as 32-bit integers from a random number generator, the distribution of k and the gcd will, for some generators, be significantly different from what we would expect from two truly independent random 32-bit integers. Note that in determining k, the number of steps in Euclid's algorithm, the first step may be 'wasted' if $u > v$—that is, the first partial quotient may be 0 and one proceeds as in versions of the algorithm where u and v are interchanged if necessary. The latter provides k's after one fewer steps. We have chosen to use the k count for unswitched u and v.

Unfortunately, we do not know the true distribution of k, the number of iterations needed, although extensive empirical study shows that k is quite close to normally distributed with mean $\mu = 18.5785$ and $\sigma = 3.405$. Since the discrete variate k is nearly normal, its distribution might be adequately represented by a member of one of the standard discrete families. Poisson will not do, since the mean is not close to the variance, but binomial might serve. The mean of the binomial distribution is Np, and the variance is Npq. Setting the ratio to $q = Npq/(Np) = \sigma^2/\mu = 3.405^2/18.5785 = .624$, say, we should consider a binomial distribution with $p = .376, q = .624$. But what value for N?.

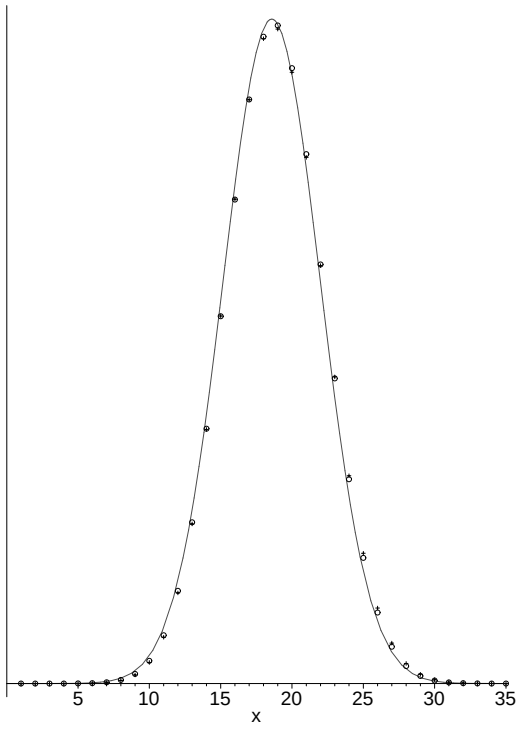

Figure 1: Points $(i, \Pr[k = i])$, binomial probabilities and the normal density, $\mu = 18.7585$, $\sigma = 3.405$

It has been known since 1733 that k, the number of steps in Euclid's algorithm, takes maximum values N when v and u are successive Fibonacci numbers—see Knuth [1], p360. This serves as a guide for possible values of N for our binomial approximation. The Fibonacci number closest to $n = 2^{32}$ is F_{48}. Experimenting with a few values for N near 48 yields the following: the distribution of k seems best fit to the binomial distribution when $N = 50$ and $p = .376$. The fit is quite good, as indicated in Figure 1, which contains the points $(i, \Pr[k = i])$, (circles), the binomial probabilities for $n = 50$, $p = .376$ (crosses), and the normal density with mean μ and standard deviation σ.

The binomial probabilities, $\Pr[k = i] = \binom{50}{i}.376^i.524^{50-i}$ serve quite satisfactorily for a test on the k values, (truncating, say, for $k \leq 3$ and $k \geq 35$), but we use a little more accurate table based on extensive simulation in our version of the gcd test.

(For the general case, we have found that for two random choices of u, v in $1, 2, \ldots, n$, the expected number of steps in Euclid's algorithm is $.842766 \ln(n) + .06535$, and the variance is $.5151 \ln(n) + .1666$.)

As for the gcd's, when u and v are chosen randomly from a range as large as 1 to 2^{32}, the distribution of their gcd is adequately close to the theoretical limit, $\Pr[\gcd = j] = c/j^2$, with $c = 6/\pi^2$. (The probability that u and v are both multiples of d is proportional to $1/d^2$, and the required probability is $\Pr[\gcd(u/d, v/d) = 1]$, given that fact. The constant comes from $1 + 1/2^2 + 1/3^2 + 1/4^2 + \cdots = \pi^2/6$.)

Note that we do not need the true distributions to develop a test of randomness. All we need do is compare the sample distribution from a particular RNG with the standard provided by a number of presumably good RNG's, 'presumably good' meaning that they produce results so close to a single one that the single one may

be used as a standard. Such a standard is based on a large number of sample distributions, each based on many repetitions (typically 10^{10} or 10^{11}) of the test procedure for RNG's that appear to give similar results. Then, if a particular RNG produces results significantly far from the standard, we say that the generator fails the test. This will usually result from a goodness-of-fit test comparison with the standard, returning a p-value very close to 0 or 1.

Use of the empirical distribution for k and the assumed iid gcd's, based on extensive simulations, leads to a test of randomness that shows some RNG's are significantly different from others when applying Euclid's algorithm to pairs of values from the generator. The most frequent disparity is in the distribution of k, the number of steps needed to terminate Euclid's algorithm, but some RNG's also differ significantly in the distribution of the gcd's.

We call this the **gcd test**. A C version is available separately, but here is an outline of its features:
Tables that provide probabilities for the expected values for the k's are included. Any k values< 3 are lumped at 3; k values>35 are lumped at 35. Probabilities for the gcd's have the form c/j^2 and need not be stored in advance, while probabilities for a gcd >100 are lumped at 100.
Tables for the cell counts for the k's and for the gcd's , `ktbl[36]` and `gcd[101]` are initialized to 0.
The following C segment generates n (usually 10^7) 32-bit pairs u and v, then increments the gcd and k tables for each pair; (`IUNI` is a #define that provides the random 32-bit integer):

```
for(i=1;i<=n;i++){k=0;
do{u=IUNI; v=IUNI;} while (u==0 || v==0); //get non-zero u and v
do{w=u%v;u=v;v=w;k++;} while(v>0);
if(u>100) u=100; gcd[u]++;
if(k<3) k=3; if(k>35) k=35; ktbl[k]++;
                  }
```

A standard goodness-of-fit test, $\sum$(obs-exp)^2/exp, compares the 33 lumped ktbl counts with the expected counts and converts to a p-value via a chi-square distribution with 32 degrees of freedom.
A standard goodness-of-fit test compares the 100 lumped gcd counts with the expected counts and converts to a p-value via a chi-square distribution with 99 degrees of freedom.

Congruential generators $x_n = ax_{n-1} \bmod p$, with p a prime and a a primitive root, consistently fail the **gcd** test, returning a p-value of 1.0000 to indicate a terrible fit to the required k distribution. So do the most commonly used of all RNG's: congruential $x_n = ax_{n-1} + \text{odd } c \bmod 2^{32}$. (For prime moduli$> 2^{32}$, the 32-bit portion is returned via a mask, while for $p = 2^{32} - 5$ the result is taken to be a random 32-bit integer.)

However, `extended congruential` generators, $x_n = a_1x_{n-1} + \cdots + a_rx_{n-r} \bmod p$ for $r > 1$ and p a prime near 2^{32} (using a 32-bit mask if $p > 2^{32}$) seem to pass both the tests: distribution of k's and distribution of the gcd's.

To look more closely at the number of steps in Euclid's algorithm for various RNG's, here is a table of the frequencies for $k \le 3$, $k = 4, \ldots, 11$ after 10^7 calls to several full-period congruential generators, (moduli 2^{32} and primes $2^{32} - 5$, $2^{32} + 15$, $2^{32} + 91$, $2^{35} + 951$) as well as for two good generators, KISS and SHR3:

k	≤ 3	4	5	6	7	8	9	10	11
Expected Counts	5.5	29.5	144.6	590.7	2065.	6277.	16797.	39965.	85157.
KISS	7	30	142	583	2148	6294	16826	39848	84943
SHR3	3	38	143	578	2098	6427	16739	40003	85388
$x_n = 69069x_{n-1}+12345 \bmod 2^{32}$	3	20	101	491	1629	4965	14076	33834	73883
$x_n = 69070x_{n-1} \bmod (2^{32}-5)$	17	36	157	633	2308	6645	17942	42515	89341
$x_n = 69071x_{n-1} \bmod (2^{32}+15)$	154	21	123	701	1896	6121	16464	39755	84924
$x_n = 69069x_{n-1} \bmod (2^{32}+91)$	142	22	140	513	1962	6314	16803	40045	84972
$x_n = 69067x_{n-1} \bmod (2^{35}+951)$	22	29	146	606	2005	6360	16707	40158	85522

For details of KISS (Keep It Simple Stupid) and SHR3, a three-shift shift register generator, see [6,7] or search 'discussions' on www.deja.com for KISS+Marsaglia, or SHR3+Marsaglia using the 'all' and 'complete' archives.

It is clear that the distribution of k, the number of steps in Euclid's algorithm for two random 32-bit integers, departs significantly from the true one, when u and v are chosen with a congruential RNG, whether with prime modulus or modulus 2^{32}. They all seem to produce a p-value very close to 1 from the standard $\sum(\text{obs-exp})^2/\text{exp}$ goodness-of-fit test.

It turns out that the average values for k are satisfactorily close to the value 18.7585 found from extensive simulation. (In our investigations, we found Knuth's expression for the expected value of k, $12\ln(2)\ln(n)/\pi^2+.06$ can be more accurately represented by $12\ln(2)\ln(n))/\pi^2+0.06535$. See Knuth[1], pages 337 and 372.)

For the distribution of the gcd of two random 32-bit integers, the results are not as striking. It is obvious that two successive values from a congruential generator $x_n = ax_{n-1} + \text{odd } c \bmod 2^{32}$ will never have an even gcd, because the trailing bit alternates. As for the congruential generators modulo a prime, most of them seem to pass the gcd distribution test, but not all of them—for example, 69070 is a primitive root for the prime $2^{32}-5$, but the congruential generator $x_n = 69070x_{n-1} \bmod 2^{32}-5$ provides an unsatisfactory distribution for the gcd of two successive outputs. Here are a few examples of expected and observed values for the gcd test.

gcd	1	2	3	4	5	6	7	8
Expected Counts	6079271	1519817	675474	379954	243171	168869	124067	94989
KISS	6078818	1521176	675496	379749	242677	168462	124215	94960
$x_n = 69067x_{n-1} \bmod 2^{35} + 951$	6077628	1520790	675940	379369	243256	168537	124310	94877
$x_n = 69070x_{n-1} \bmod 2^{32} - 5$	5707365	1428317	973541	357324	228086	243726	117013	89016
$x_n = 69069x_{n-1}+12345 \bmod 2^{32}$	8106215	0	900376	0	324000	0	165701	0

3 The Gorilla Test

Merits of this test are suggested by a quote from reference [4]:

> 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.

The idea is to use the random number generator to produce a sequence of 'letters' from an alphabet, then study the frequency of k-letter words in that sequence. These were called **monkey tests** in [4], and they account for many of the most-difficult-to-pass tests in the Diehard battery. Making the sequences very long serves to show the shortcomings of many RNGs, particularly those whose relatively short periods make them unable to produce a reasonable proportion of the possible k-tuples.

Probably the best way to assess the distribution of k-letter words produced by the 'monkey' is to create a cell for each possible word, then count the number of appearances of each word in a long sequence of letters produced by the monkey. Then the quadratic form in a weak inverse of the covariance matrix of the zero-adjusted cell counts provides a chi square test. That turns out to be $Q_k - Q_{k-1}$, where Q_k is the naive Pearson quadratic form, $\sum(\text{observed-expected})^2/\text{expected}$, for words of length k, see Marsaglia [3]. Unfortunately, there are usually too many possible k-letter words to keep cell counts for each one. An alternative, as advocated in [3], is to count the number of missing k-letter words in a long string produced by the monkey. The resulting count should be approximately normally distributed with mean determined by theory and variance determined by extensive simulation.

That is the basis of the test included here. As a strong version of a monkey test, we call it the **gorilla test**. It is applied as follows:
For 32-bit integers produced by the generator, specify one of the 32 possible bit positions, with the bits numbered 0 to 31 from most- to least- significant. Form a sequence of $2^{26} + 25$ such bits, made up of the designated bit of each of $2^{26} + 25$ integers from the generator. If x is the number of 26-bit 'words' that do not appear in that sequence, then x should be approximately normally distributed with mean 24687971 and standard deviation 4170, so that $\Phi((x - 24687971)/4170)$ should be uniformly distributed in $[0, 1)$, that is, provide a p-value for the test, where $\Phi()$ is the standard normal distribution function.

Below are examples of the output of the Gorilla Test for various RNG's. The output contains the p-value for each of the 32 bit positions of the generator's output. Recall that we first specify the bit position to be studied, from 0 to 31, say bit 3. Then we generate 67,108,889 ($2^{26} + 25$) numbers from the generator and form a string of 67,108,889 bits by taking bit 3 from each of the generator's numbers. In that string of 67,108,889 bits we count the number of 26-bit strings that do not appear. That count should be approximately normal with mean 24687971 and standard deviation 4170. Converting this by means of the appropriate normal probability transformation, we get a number between 0 and 1, a p-value. The output of the Gorilla Test gives the p-values for each of the 32 bit positions, then does an Anderson-Darling-Kolmogorov-Smirnov test on those 32 presumed uniform (0,1) values.

```
Gorilla Test for KISS
Bits  0 to  7 || 0.6330 0.2903 0.6350 0.7377 0.1342 0.6095 0.1959 0.3699
Bits  8 to 15 || 0.4194 0.9699 0.3807 0.4496 0.9106 0.9100 0.4753 0.8187
Bits 16 to 23 || 0.3225 0.2455 0.7300 0.9907 0.0483 0.8786 0.3932 0.9093
Bits 24 to 31 || 0.0975 0.2096 0.5962 0.3991 0.2822 0.4591 0.6845 0.1816
  0.115=ADKS
Gorilla Test for SHR3: jsrˆ=(jsr<<13); jsrˆ=(jsr>>17); jsrˆ=(jsr<<5);
Bits  0 to  7 || 0.1301 0.9562 1.0000 0.5639 0.3534 0.5053 0.5459 0.7153
Bits  8 to 15 || 0.6989 0.5975 0.5579 0.2299 0.4949 0.4399 0.3033 0.2713
Bits 16 to 23 || 0.4054 0.0514 0.9929 0.5981 0.6724 0.3801 0.2743 0.0367
Bits 24 to 31 || 0.5239 0.5100 0.1128 0.8865 0.8057 0.8623 0.9569 0.0000
    0.937=ADKS
Gorilla Test for LFIB4: x(n)=x(n-256)+x(n-179)+x(n-119)+x(n-55) mod 2ˆ32.
Bits  0 to  7 || 0.7726 0.6625 0.8484 0.6311 0.5161 0.4235 0.3163 0.0502
Bits  8 to 15 || 0.0928 0.6614 0.0078 0.2021 0.6616 0.0149 0.5762 0.5736
Bits 16 to 23 || 0.4923 0.6725 0.5489 0.1335 0.8364 0.2657 0.0169 0.7038
Bits 24 to 31 || 0.5774 0.7989 0.6508 0.4192 0.2158 0.6698 0.8185 0.2468
    0.724=ADKS
Gorilla Test for x(n)=69069*x(n-1) mod 2ˆ32+91
Bits  0 to  7 || 0.0309 0.0211 0.0260 0.0150 0.0181 0.0002 0.0202 0.0162
Bits  8 to 15 || 0.3523 0.0018 0.0013 0.4848 0.0345 0.0044 0.0008 0.0076
Bits 16 to 23 || 0.0227 0.0012 0.0018 0.0743 0.0821 0.0001 0.0118 0.4472
Bits 24 to 31 || 0.3204 0.5452 0.3325 0.4709 0.7966 0.5493 0.0068 0.3143
    1.000=ADKS
Gorilla Tests for x(n)= 214013*x+2531011 mod 2ˆ32 (gcc's rand function)
Bits  0 to  7 || 0.6429 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000
Bits  8 to 15 || 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Bits 16 to 23 || 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Bits 24 to 31 || 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
   1.000=ADKS
```

Some generators will fail the Gorilla Test at some particular bit position, as does SHR3 at bits 2 and 31, or may fail because collectively their p-values do not seem uniform in (0,1), as does the congruential generator with prime modulus $2^{32}+91$, with far too many small p-values.

The Gorilla Test takes quite a bit longer than the other tests described here.

4 The Birthday Spacings Test

The birthday spacings test is an extension of the iterated spacings test described by Marsaglia in [3]. If m birthdays are randomly chosen from a year of n days, then sorted, the number of duplicated values among the spacings between those ordered birthdays will be asymptotically Poisson distributed with parameter $\lambda = m^3/(4n)$. Theory provides little guidance on speed of the approach to limiting form, but extensive simulation with a variety of RNG's provides values of m and n for which the limiting Poisson distribution seems satisfactory. Among these is $m = 1024$ birthdays for a year of length $n = 2^{24}$ with $\lambda = 16$, the version used in Diehard [6], and the more stringent versions, $n = 2^{20}, 2^{23}, 2^{26}, 2^{29}, 2^{32}$; $m = 256, 512, 1024, 2048, 4096$.

Experience suggests that a RNG that passes the birthday spacing test for $n = 2^{32}, m = 2^{12}, \lambda = 4$ will pass the test for those other values of n, m, λ, so we have chosen $n = 2^{32}$ and $m = 2^{12}, \lambda = 4$, as the most-difficult-to-pass version of the birthday spacings test. That is the one used here. We shorten the name to **the bday test**. It goes as follows: The RNG in question produces 4096 birthdays. Each birthday is a 32-bit integer, a 'day' in a 'year' of 2^{32} days. The 4096 birthdays are sorted, then differences taken to get the spacings. Then the spacings are sorted, from which a count of the number of duplicate spacings is evident. That provides a purported Poisson value with mean $\lambda = 4$. This process is repeated to get a sample of 5000 from that Poisson distribution. Then a goodness-of-fit test, $\sum(\text{observed- expected})^2/\text{expected}$, is applied to the Poisson sample, the result inserted into the appropriate chi-square distribution function to get a value in $[0, 1)$—a p-value.

Here is an example of output from the bday test for a good RNG:

```
          Table of Expected vs. Observed counts:
Generator: KISS  (Combines congruential, shift-register and multiply-with-carry)
  0      1      2      3      4      5      6      7      8      9   >=10
 91.6  366.3  732.6  976.8  976.8  781.5  521.0  297.7  148.9   66.2   40.7
  87    385    748    962    975    813    472    308    159     61     30
          Chi-square p for that table:  0.705
```

And here are outputs for several generators that fail the bday test:

```
Generator: x(n)=214013*x(n-1)+2531011 mod 2^32     (The gnu rand() function)
  0      1      2      3      4      5      6      7      8      9   >=10
 91.6  366.3  732.6  976.8  976.8  781.5  521.0  297.7  148.9   66.2   40.7
2112   1797    815    221     46      8      0      1      0      0      0
        Chi-square p for that table:  1.000
```

```
Generator: x(n)=x(n-856)-x(n-920)-borrow mod 2^32 (subtract-with-borrow, period>10^8859)
  91.6  366.3  732.6  976.8  976.8  781.5  521.0  297.7  148.9   66.2   40.7
   42    197    482    815    902    824    724    462    275    159    118
        Chi-square p for that sample:  1.000
```

```
Generator: x(n)=x(n-55)+x(n-24) mod 2^32  (A Knuth favorite)
   0      1      2      3      4      5      6      7      8      9   >=10
 91.6  366.3  732.6  976.8  976.8  781.5  521.0  297.7  148.9   66.2   40.7
  15     60     255    470    691    837    814    666    489    319    384
        Chi-square p for that sample:  1.000
```

Notice the different patterns of the observed versus expected values for generators that fail the birthday spacings tests. For some generators, there are far too many values less than the mean, 4, while for others there are far too many values greater than 4.

5 Summary and Remarks

We have presented three tests: **gcd, Gorilla** and **bday**.
The first test, gcd, tests whether k, the number of steps to completion of Euclid's algorithm, and the resulting gcd, both have the distributions called for by the underlying probability theory for two independent uniform 32-bit random integers. Many RNG's fail this test; in particular congruential RNG's—even those with a prime modulus—seem to fail the test on the distribution of k, and often on the distribution of the gcd. An analogous test could be used for RNG's producing more than 32 bits; the gcd's should take value j with frequency proportional to $1/j^2$, but the distribution of k, steps to completion, may require a different table of probabilities.

Failure of a RNG to satisfy randomness aspects of Euclid's algorithm, the oldest and one of the most fundamental of those in number theory, may be of concern to those using procedures in computational number theory where conclusions are based on the assumption that random selections of integers from 1 to n are indeed independent and uniform. Examples are in probable prime tests or the complexity of factoring algorithms that depend on random selections.

The second test, the Gorilla Test, is based on what we have called Monkey Tests, in the sense of the RNG as a monkey at a typewriter, randomly producing a very long string of keystrokes. The number of missing k-letter words should be approximately normal with certain mean and variance depending on word size, alphabet size and probabilities for letters of the alphabet. A particularly strong monkey test, hence the Gorilla Test, chooses a particular bit from each 32-bit random integer, forms a string of 2^{26} such bits, counts the number of missing 26-bit 'words' in that long string. The test is applied to each of the 32 bit positions; surprisingly, some RNG's consistently fail for certain bit positions.

An analogous version could be used for generators producing fewer or more than 32 bits.

The third test, bday, is a strong version of Marsaglia's Birthday Spacings Test, in which each random integer is used to determine a birthday in a year of n days. With m birthdays chosen and put in increasing order, the number of duplicate values among the spacings between the birthdays should have a Poisson distribution with parameter $\lambda = m^3/(4n)$. The version here is strong in the sense that many RNG's might do well for smaller m's and n's, but fail as one or the other is increased. A year of $n = 2^{32}$ days is the implicit limit for 32-bit RNG's, while choosing $m = 4096$ birthdays seems to provide a test that some otherwise promising RNG's fail.

Analogous versions could be applied to RNG's on 48, 64 bits, etc, with n increased. Applications where tests such as bday may have implications are in computational number theory, where many procedures use a selection of random integers from 1 to n and conclusions are based on the assumption that they are independent and uniform. Of course, such applications often involve random integers with hundreds of bits, but experience with 32 bits suggests that analogous tests for larger integers may be worth considering.

6 References

[1] Knuth, Donald E. (1998) *The Art of Computer Programming, Volume II*, 3rd Ed., Addison Wesley, Reading, Mass.

[2] MacLaren, D. and Marsaglia, G., (1965), Uniform random number generators,", *Journ. Assoc. for Computing Machinery*, **12**, 83–89.

[3] Marsaglia. G.,(1985), A current view of random number generators, Keynote Address, Statistics and Computer Science: XVI Symposium on the Interface, Atlanta, *Proceedings*, Elsevier.

[4] Marsaglia, G. and Zaman, A., (1995), Monkey tests for random number generators, *Computers & Mathematics with Applications*, **9**, No. 9, 1–10.

[5] Marsaglia, G. and Zaman, A., (1995), Some very-long-period portable random number generators, *Computers in Physics*, **8** 117–121.

[6] The Marsaglia Random Number CDROM, with The Diehard Battery of Tests of Randomness, produced at Florida State University under a grant from The National Science Foundation, 1985. Access available at www.stat.fsu.edu/pub/diehard.

[7] Marsaglia, G. Random Number Generators for C: Some suggestions. Postings in newsgroups sci.math, sci.math.numer-analysis, sci.stat.math, Jan 1999. Full thread with responses available via deja.com.