Tuning the Collision Test for Power

W.W.Tsang, L.C.K. Hui, K.P. Chow, C.F. Chong, and C.W. Tso

Department of Computer Science and Information Systems
The University of Hong Kong

Email: tsang@csis.hku.hk

Abstract

The collision test is an important statistical test for rejecting poor random number generators. The test simulates the throwing of balls randomly into urns. A problem in applying this test is to determine the number of urns, *m*, and the number of balls, *n*, so that the test is among the most powerful possible on a computer.

The problem was tackled empirically. A set of canonical congruential generators with increasing periods was first implemented. The stringency of a test against congruential generators is measured as the number of canonical generators the test rejects. Experiments were then conducted to measure the stringencies of the collision tests for various (*m*, *n*) values. The results reveal that for a fixed *m*, the stringency of a test reaches maximum when $m < n \leq 2m$. Moreover, the stringency increases as *m* increases. Similar results were observed when the experiments were repeated on lagged Fibonacci generators and on shift-register generators. Further investigation showed that the variance of the number of collisions reaches maximum together with the stringencies, at $n = \lfloor 1.256431m \rfloor$.

Eighteen well-known generators were tested against the collision tests with $m = 2^{21}, 2^{22}, \ldots,$ up to 2^{30}. Many generators failed starting from some points along the way, including congruential generators, shift-register generators, lagged Fibonacci generators of lags less than 40, subtract-with-borrow generators of lags less than 24, and a combined generator.

Keywords: Random number testing, Statistical tests, Collision test

*Research supported by Innovation and Technology Support Programme, HKSAR, Grant ITS/277/00, and the Hong Kong Research Grants Council (RGC), Grant HKU 7046/02E.

1 Introduction

The collision test suggested by H. Delgas Christiansen in 1975 is among the foremost statistical tests for rejecting poor random number generators (RNGs). The test simulates throwing balls randomly into urns. Let *m* be the number of urns and *n* be the number of balls thrown. *m* is usually a power of 2 and the destination of a ball is determined by $\log_2 m$ bits produced by the generator being tested. When a ball falls into an urn that is already occupied, a *collision* occurs. The collision test counts the number of collisions, *c*. An RNG fails in the test if *c* falls outside a predefined interval. The test requires *m* bits in RAM to keep track of the statuses of urns. The run-time complexity of the test is O(*n*).

The collision test is important because throwing balls is identical to insertion of items into a hash table and collisions are the major concern in both cases. It is one of a handful of statistical tests for RNGs that are highly recommended by D. Knuth. A comprehensive description of the test, with an example that throws $n = 2^{14}$ balls into $m = 2^{20}$ urns, was included in his classic book [Knuth 1997]. Since then, the collision test with these specific values for *m* and *n* was used to test RNGs [Vattulainen 1995]. A problem in applying the test is whether $n = 2^{14}$ is a good choice for $m = 2^{20}$ or not. Will the test become more *powerful*, i.e., with higher ability in rejecting bad generators, if $n = 2^{10}$ or $n = 2^{18}$? In general, how shall we determine the values of *m* and *n* so that the test reaches its highest power on a computer available for testing? Can the power of the test be scaled up when more RAM and more powerful CPU become available in the future?

Our studies showed that for a fixed *m*, *n* shall be chosen equal to $\lfloor 1.256431m \rfloor$. The power of the test with *n* determined this way increases as *m* increases. This conclusion was reached by estimating the power of the collision tests of various (*m*, *n*) values against three families of RNGs, namely, congruential, lagged Fibonacci, and shift-register generators. For each family, we first implemented a set of canonical generators of periods of different magnitudes. The *stringency* of a test against the family is defined as the number of the canonical generators the test rejects. With this tool, we found that for a fixed *m*, when *n* increases, the stringency increases but eventually levels off or drops down. The maximum stringency first occurs when $m < n \leq 2m$. The same phenomenon was observed in all three families of RNGs. Further investigation showed that *n* should be determined subject to maximizing the variance of c, asymptotically at $n = \lfloor 1.256431m \rfloor$.

The collision test of $n = \lfloor 1.256431m \rfloor$, with scalable m was implemented. As expected, the stringency of the test increases when m increases. A generator being examined was tried out with this test with $m = 2^{21}$, 2^{22}, …, up to 2^{30}. Many well-known generators of various kinds were rejected starting from certain points along the way. Three congruential generators with modulus equal to 2^{32} failed when $m \geq 2^{24}$. Two with modulus equal to $2^{31}-1$ sustained the tests better but nonetheless failed when $m \geq 2^{26}$. One with modulus equal to 2^{48} failed when $m \geq 2^{28}$. Two shift-register generators [Golomb 1982] failed when $m \geq 2^{23}$. The lagged Fibonacci generators generally passed but those with lags less than 40 failed. Similar results were obtained for the subtract-with-borrow generators [Marsaglia 1991]. The least significant bits of words generated by Super-Duper, a combination generator, failed as early as $m = 2^{23}$. The Mersenne Twister [Matsumoto 1998] passed alright, so did the KISS generator [Marsaglia 1999].

One major difficulty we encountered in our investigation was computing the distribution of c. D. Knuth has suggested a recursive procedure that gives exact values. The procedure works well when n is much smaller than m but takes too long to compute when m is large and n is close to m. To cope with the latter cases, we compute the normal approximation of the distribution of c instead of the exact one. Such approach was adopted in working out the statistic in the monkey tests [Marsaglia 1993]. The variance of the statistic there is estimated using simulation, whereas the variance of c here can be computed using the exact formula we derived.

An analysis on the Knuth's procedure for computing the distribution of c is presented in Section 2. When m is large and n equals m, the run-time complexity of the method is found to be O($n^{3/2}$). The formulas for the mean and the variance of c are derived in Section 3. The accuracy of the normal approximation to the exact distribution of c is assessed there. In Section 4, we give the details of our pursuit in determining n subject to maximizing the stringencies. We devise the maximum variance criterion, and explain how to reach the conclusion that n approaches $\lfloor 1.256431m \rfloor$ asymptotically. Finally, we applied the fine tuned collision tests of scalable m on many generators. The test results are presented in Section 5.

2 Distribution of the number of collisions

Consider throwing n balls randomly into m urns. The probability that c collisions occurs in a collision test is

$$\frac{m(m-1)\cdots(m-n+c+1)}{m^n}\left\{ {n \atop n-c} \right\},$$ where $\left\{ {n \atop k} \right\}$ is a Sterling number of 2nd kind defined as $\left\{ {n \atop 1} \right\} = 1$, $\left\{ {n \atop n} \right\} = 1$,

$$\left\{ {n \atop k} \right\} = k\left\{ {n-1 \atop k} \right\} + \left\{ {n-1 \atop k-1} \right\}$$ otherwise [Knuth 1997].

Based on a recursion derived from the above formulas, D. Knuth has given an algorithm for computing the percentiles of collisions. The function *pcoll1()* shown in Figure 1 is a C implementation of the Knuth's algorithm that computes the cumulative probability of c collisions.

```
double pcoll1(int m, int n, int c)       /* Compute the cdf of c collisions */
{               double *A, mm, cdf;
                int i, j, j0, j1;

                mm = m;
                A = (double *) malloc( (n+1) * sizeof(double));

                for (j=0; j<=n; ++j)                          /* S1 */
                   A[j] = 0.;
                A[1] = 1.;

                j0 = 1; j1 = 1;

                for (i=1; i<n; ++i)                                         /* S2 */
                {   j1 = j1 + 1;
                   for (j=j1; j>=j0; --j)
                       A[j] = (j/mm) * A[j] + ((1.+ (1./mm))-(j/mm)) * A[j-1];

                   if (A[j0] < 1e-20) A[j0++] = 0.;
                   if (A[j1] < 1e-20) A[j1--] = 0.;
                }

                if (n-c > j1) {free(A); return 0.;}           /* Compute the cdf */
                if (n-c < j0) {free(A); return 1.;}

                cdf = A[j1];
                while (n-c < j1)
                                cdf = cdf + A[--j1];
                free(A);
                return cdf;
}
```

Figure 1. A C function that computes the CDF of c using the Knuth's method.

The execution time of *pcoll1()* is proportional to the product of n and the number of non-zero entries in array A[] (An entry with value less than 10^{-20} is immediately set to zero in the program). As the distribution approximates to normal (established in the next Section) and a normal density has most of its underneath area within a few standard deviations from the mean, the number of entries with non-zero values in A[] is of order of the standard deviation, i.e., the square root of the variance of c. Using MAPLE, the Taylor expansion of Formula (3.1) for the variance of c given in the next Section are

$$\frac{1}{24m^3}\begin{pmatrix} 12n^2m^2 - 12nm^2 - 20n^3m + 48n^2m \\ -28nm + 17n^4 - 78n^3 + 115n^2 - 54n \end{pmatrix} + O\left(\frac{1}{m^4}\right).$$

When $n = m$, the variance is of order n. Therefore, the number of non-zero entries in A[] is of order $\sqrt{n}$. Consequently, the run-time complexity of *pcoll1()* is O($n^{3/2}$). Figure 2 shows the execution times of *pcoll1()* against $\log_2 n$, for $m = 2^{20}$. The times (in seconds) were measured on a PC with a 450 MHz CPU. When $n = m \geq 2^{21}$, *pcoll1()* will take too long to complete and a more efficient method is needed.

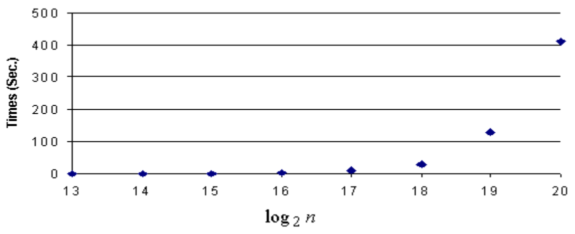


Figure 2. The execution time of *pcoll1*() versus $\log_2 n$, for $m = 2^{20}$.

3 Normal approximation

In this section, we derive an approximation to the distribution of *c* from the occupancy problem which concerns with the number of empty urns, *e*. The collision test and the occupancy problem are indeed the two sides of a coin. The relation between *e* and *c* is *e* = *m* – *n* + *c*.

A thorough discussion on the classical occupancy problem was included in W. Feller’s classic book [Feller 1950]. A theorem due to von Mises states that *e* is approximately Poisson distributed with mean, $\lambda = me^{-n/m}$, under the conditions that *m* and *n* are large and that λ remains bounded. As a Poisson distribution approximates to normal when λ increases, *e* asymptotically follows the normal distribution with both the mean and variance equal to λ. For *m* and *n* that are not excessively large, the approximation will be better if the variance of the normal are set to the exact variance of *e*, σ_e^2, instead of λ [Marsaglia 1993].

The mean and variance of *e* can be worked out from the occupancy of the urns. Suppose that we throw *n* balls randomly into *m* urns. For *i* = 1 to *m*, let

$$X_i = \begin{cases} 1, & \text{if urn } i \text{ is empty;} \\ 0, & \text{otherwise (occupied).} \end{cases}$$

The probability that a ball hits a particular urn is *1/m*. The probability that it misses is $1-1/m$. The probability that the urn is empty, i.e., all *n* balls miss the cell, is $q = (1-\frac{1}{m})^n$. Moreover,

$$E(X_i) = q,$$
$$Var(X_i) = E(X_i^2) - E(X_i)^2 = q - q^2.$$

Next, consider the occupancy of two particular cells, *i* and *j*, where $i \neq j$. The probability that the first ball does not hit both cells is $(m-2)/m$. The probability that both cells are empty, i.e., all balls miss both cells, is $r = (1-\frac{2}{m})^n$. The covariance of X_i and X_j is

$$Cov(X_i, X_j) = E(X_i X_j) - E(X_i)E(X_j) = r - q^2.$$

Since $e = X_1 + X_2 + \cdots + X_m$,

$\mu_e = mE(X_1) = mq$. Furthermore,

$$\sigma_e^2 = Var(X_1) + \cdots + Var(X_m) + \sum_i \sum_{j \neq i} Cov(X_i, X_j)$$
$$= mVar(X_1) + (m^2 - m)Cov(X_1, X_2)$$
$$= m(q - q^2) + (m^2 - m)(r - q^2)$$
$$= m(q + mr - r - mq^2)$$

As *c* = *e* – *m* + *n*, *c* is approximately normal distributed with

$\mu_c = \mu_e - m + n = mq - m + n$, and

$$\sigma_c^2 = \sigma_e^2 = m(q + mr - r - mq^2). \quad (3.1)$$

The closeness of the distributions of *c* to normal is demonstrated in Figure 3. The histograms of the exact distributions and their corresponding normal densities were plotted together for various values of *m* and *n*. In general, the approximation becomes better when *m* and *n* increase.

A simple C function, *pcoll2()*, which computes the cumulative distribution of *c* using the normal approximation is given in Figure 4. Comparing with *pcoll1()*, *pcoll2()* is fast but less accurate when *m* or *n* is small. We may use it to replace *pcoll1()* when both *m* and *n* are larger than 2^{16}. For $m \geq 2^{17}$ and $n \leq m$, the largest absolute error in the values returned by *pcoll2()* that are less than 0.05 or over 0.95 (regions possibly leading to rejections of hypotheses) is 0.000446. This upper limit of error occurs when $m = 2^{17}$, $n = 2^{17}$ and *c* = 48404. Such accuracy is acceptable in most applications.

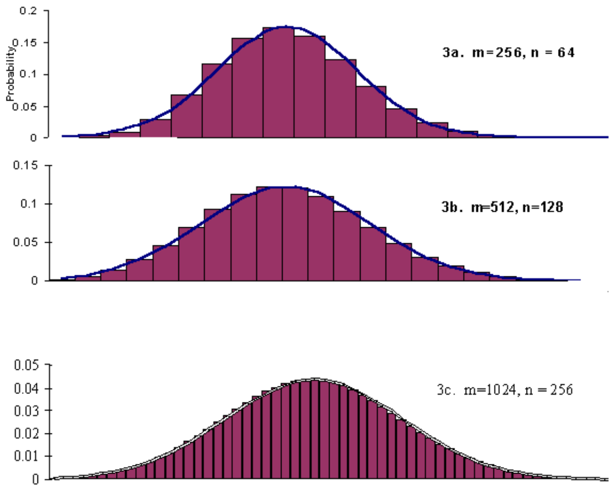


Figure 3. The exact distributions of *c* and their normal approximations.

```
double pcoll2(int m, int n, int c)
{           double mm, q, r, mean, var;

            mm = m;
            q = exp(n * log(1.-1./mm));
            r = exp(n * log(1.-2./mm));
            mean = mm * q - mm + n ;
            var = mm * (q + mm*r - r - mm * q * q);

            /* Phi() computes the cdf of standard normal */
            return Phi( (c-mean)/sqrt( var));
}
```

Figure 4. A C function that computes the CDF of *c* using the normal approximation

4 How many balls shall be thrown

One interesting problem of the collision test is for a fixed number of urns, how many balls shall be thrown. Throwing excessively too few balls is not likely to have collisions at all. On the other hand, throwing too many balls takes longer to complete but does not necessarily lead to a more powerful test. So, what will be a good choice for the number of balls?

As the collision test is used to examine RNGs, we would like to choose *n* such that the test has the highest ability in rejecting poor generators. Such ability is referred to as *power* in statistics. Its formal definition is the probability of not making Type II error, i.e., the probability of rejecting poor generators. The power of a test against a particular deterministic generator can be estimated by conducting the test using different seeds. The power of a test against a family of generators is a complicated discrete function and has rarely been studied. Not to mention, the power of a test in general is merely a qualitative concept. No one really knows how to measure it. However, if we want to tune a test for maximum power, we need to be able to make the measurement. To overcome this difficulty, we first established a practical way that measures the *stringency* of a test against congruential generators. We assume that the stringency so defined is positively correlated to the power of the test against the generators in the family. This assumption was verified by numerous experiments. We then tuned the collision test for maximum stringency. The same experiment were repeated on lagged Fibonacci generators, and finally on shift-register generators. Results from experiments reveal that the test reaches maximum power against all the three families of RNGs when $m < n \leq 2m$.

According to Equation 3.1, for a fixed *m*, the variance of *c* is a bell-shaped function of *n*. An interesting discovery is that the location of the maximum of this function coincides with the maximum of the power. Thus, *n* shall be determined subject to maximizing the variance of *c*. With this *maximum variance criterion*, we found that asymptotically, $n = \lfloor 1.256431m \rfloor$. Details of our investigation are described below.

4.1 Tuning the test against linear congruential generators

Linear congruential generator is the most well studied and popular deterministic generator. It was proposed by Lehmer in 1949 [Lehmer49] and is among the fastest RNGs. The general formula is $X_{i+1} = a\, X_i + b \bmod k$. As *b* is non-essential, let us focus on multiplicative linear congruential generator (MLCG) where $b = 0$. The power of a test against a particular MLCG can be estimated using different seeds. The power of a test over the family of MLCGs, however, is a complicated discrete functions on *a* and *k* and is formidable to deal with. Instead, we make up a heuristic index called stringency that is positively correlated to the expected height of the power function. A test is then tuned for maximum stringency.

First, we implemented 29 canonical MLCGs, with periods varying from roughly 2^{16} to 2^{44}. In these generators, we adopted the parameters (*a*, *k*)'s published in [L′Ecuyer 1999]. These parameters were chosen such that the resulting generators perform well in the spectral test [Knuth 1997]. Table 5 shows the actual values of these parameters of the canonical MLCGs. To gauge the stringency of a test, we conduct the test on the canonical generators one by one, from the shortest period to the longest. The *stringency* of the test is defined as the number of generators it rejects, before the first generator it passes.

	a	*k*
1	2469	$2^{16}-15$
2	29803	$2^{17}-1$
3	21876	2^{18}-5
4	155411	$2^{19}-1$
5	22202	$2^{20}-3$
6	1939807	$2^{21}-9$
7	1731287	$2^{22}-3$
8	422527	$2^{23}-15$
9	931724	$2^{24}-3$
10	25612572	$2^{25}-39$
11	66117721	$2^{26}-5$
12	3162696	$2^{27}-39$
13	104122896	$2^{28}-57$
14	530877178	$2^{29}-3$
15	921746065	$2^{30}-35$
16	784588716	$2^{31}-1$
17	279470273	$2^{32}-5$
18	7312638624	$2^{33}-9$
19	473186378	$2^{34}-41$
20	8094871968	$2^{35}-31$
21	45453986995	$2^{36}-5$
22	85876534675	$2^{37}-25$
23	24271817484	2^{38}-45
24	541240737696	2^{39}-7
25	937333352873	2^{40}-87
26	1319743354064	2^{41}-21
27	92644101553	2^{42}-11
28	3663455557440	2^{43}-57
29	949305806524	2^{44}-17

Table 5. The values of the (*a*,*k*)'s of the canonical MLCGs.

As an example, we have applied the collision test of $m = 2^{20}$ and $n = 2^{21}$ to the canonical generators. Only the most significant bit (MSB) of a number generated is used in the test. The number of collisions counted was converted into a p-value. Figure 6 shows the resulting p-values. The test rejects a generator when $U < 0.001$ or $U > 0.999$. The first 10 generators failed in the test and the rest passed. According to the definition, the stringency of this collision test is 10.

```
Collision Test m=2^20, n=2^21
 -----------------------------
 U:<mlcg16>..<mlcg23>| 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 0.0002
 U:<mlcg24>..<mlcg31>| 0.0000 0.0000 0.0840 0.0290 0.2367 0.6886 0.0618 0.0155
 U:<mlcg32>..<mlcg39>| 0.3846 0.7827 0.8937 0.4450 0.9727 0.9402 0.9438 0.1707
 U:<mlcg40>..<mlcg44>| 0.1139 0.6898 0.2345 0.4112 0.4341

 The stringent level against MLCG is 10
```

Figure 6. The results of applying the collision test with $m = 2^{20}$ and $n = 2^{21}$ to the canonical MLCGs.

Figure 7 shows the stringencies of the collision tests with $m = 2^{20}$ and $n = 2^{10}, 2^{11}, \ldots, 2^{28}$. The stringency increases as n increases until $n = 2^{20}$. Thereafter, the stringency remains more or less constant. To understand such behaviour, we superimposed the curve of the variance of c in the bar chart. (A variance is computed for each test using Equation 3.1. These values are connected together with a smooth curve. The scale of the curve and the bars are different.). When n is very small, c tends to zero with small variance. Such a test can hardly tell whether a generator is good or bad and its stringency is low. When n increases, the variance increases, and the stringency increases too. Our explanation is that large variance of c provides more room for a bad generator to be bias and therefore leads to higher chance of rejecting the generator. As the variance drops when n increases beyond the abscissa of the maximum variance, n_v, we anticipated that the stringency drops along. This is however not the case—the stringency remains at high level and forms a plateau. It is because in the testing of congruential generators with short periods, some urns remain empty no matter how many balls have been thrown.

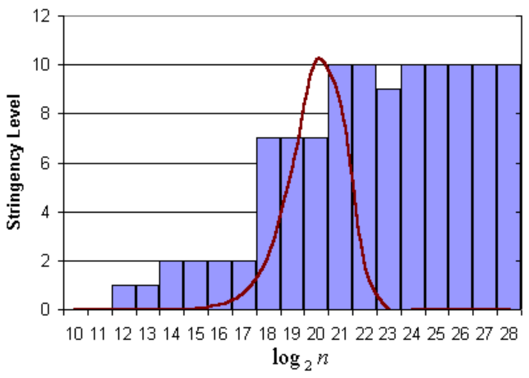


Figure 7. The stringencies against MLCG and variance of collision tests of $m = 2^{20}$.

From the above results, we might not want to choose n beyond n_v. It is because additional computational effort is needed but no stringency is gained. In other words, n should be determined subject to maximizing the variance of c. We call this the *maximum variance criterion*. But what is the value of n_v? Using the numerical methods in *Maple*, we found that $n_v = 1.25643088m$, $1.25643119m$ and $1.25643121m$ for $m = 2^{20}$, 2^{24}, and 2^{29} respectively. Therefore, we suggest to choose n equal to $\lfloor 1.256431m \rfloor$ in the collision test for $m \geq 2^{20}$.

4.2 Tuning the test against lagged Fibonacci generators

Lagged Fibonacci generator (LFG) was originally suggested by Mitchell and Moore in 1958. The formula is $X_n = X_{n-p+q} + X_{n-p} \bmod m$. In a 32-bit computer, m is usually set to 2^{32} for efficiency. The indices p and q are chosen such that $x^p + x^q + 1$ is a primitive trinomial. The period of such a generator is $2^{31}(2^p - 1)$. This generator is fast and easy-to-implement. The most prominent feature is that a very long period can be achieved by choosing a large p.

Twenty seven canonical LFGs with periods of various magnitudes were implemented. The parameters, (p, q)'s were taken from a table published on page 28 in [Knuth 1981]. They are chosen such that the resulting generators are not too difficult to fail. Table 8 shows the values of these parameters. The stringency of a test against LFGs is defined in the same ways as MLCGs. The measurement is also similar except that the least significant bit (LSB), instead of the MSB, of a number generated is used in the testing. It is because the MSB sequences of LFGs are too difficult to fail. Figure 9 shows the stringencies of the collision tests with $m = 2^{20}$ and $n = 2^{10}, 2^{11}, \ldots, 2^{28}$. The stringency increases as n increases until $n = 2^{20}$. Thereafter, the stringency drops down. As in the graph of MLCGs, the location of the maximum stringency coincides with the maximum of the variance of c.

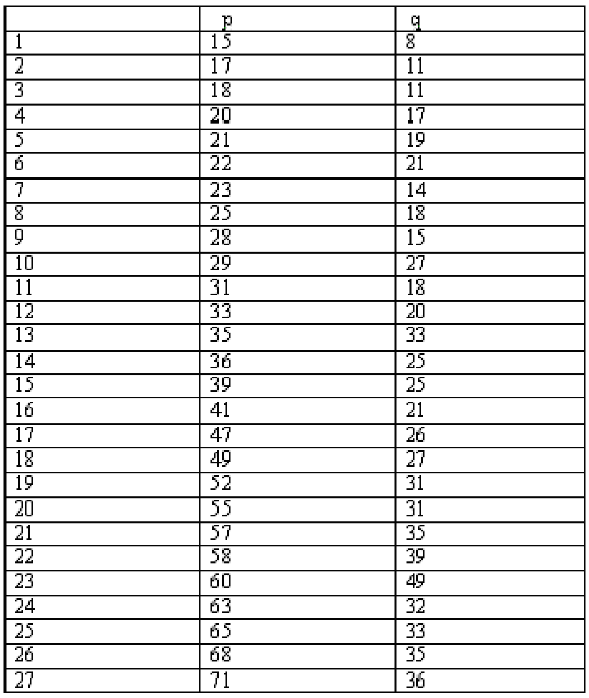
4.3 Tuning the test against shift-register generators

A shift-register generator computes the next number using the left-shift, right-shift and bitwise exclusive-or operations [Golomb 1982]. Consider a number, w, as a bit sequence. The function left-shift(w, l) shifts w to the left l bits. right-shift(w, r) shifts w to the right r bits. exclusive-or(u, v) conducts a bitwise exclusive-or of u and v. Suppose that the current number of a shift-register generator of parameters (l, r) is W. The next number is computed by the following.

W = exclusive-or(W, left-shift(W, l));

W = exclusive-or(W, right-shift(W, r));

If W is treated as a binary vector, the overall computation can be done by multiplying W with a binary matrix [Marsaglia 1985]. The parameters (l, r) are chosen such that the matrix has order $2^{|W|} - 1$ in the group of nonsingular $|W| \times |W|$ binary matrices. The theory is also applicable to three-shift generators (SHR3) that have an additional round of left-shift and exclusive-or after the first two rounds [Marsaglia 1999]. SHR3 has the same maximum period, $2^{|W|} - 1$, as the two-shift version but scrambles the bits more thoroughly.



	p	q
1	15	8
2	17	11
3	18	11
4	20	17
5	21	19
6	22	21
7	23	14
8	25	18
9	28	15
10	29	27
11	31	18
12	33	20
13	35	33
14	36	25
15	39	25
16	41	21
17	47	26
18	49	27
19	52	31
20	55	31
21	57	35
22	58	39
23	60	49
24	63	32
25	65	33
26	68	35
27	71	36

Table 8. The values of the (*p*, *q*)'s of the canonical LFGs.

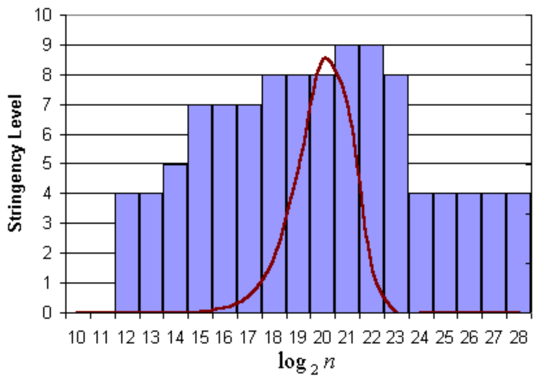
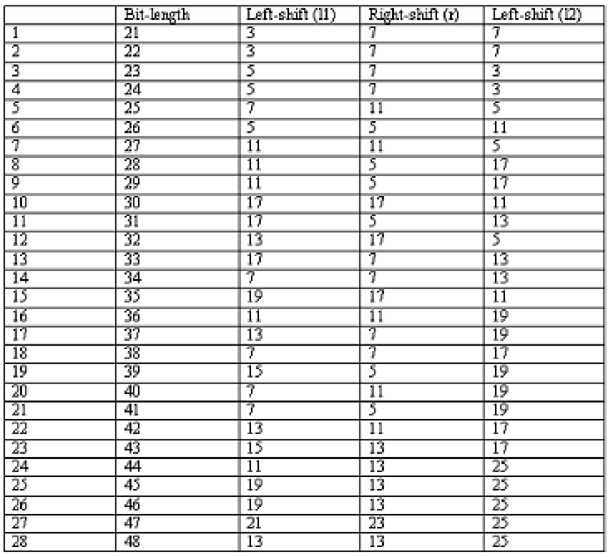


Figure 9. The stringencies against LFG and variance of collision tests of $m = 2^{20}$.

Twenty eight canonical SHR3s of various bit lengths were implemented. The parameters, (*l1*, *r*, *l2*)'s, were found using a Maple program that assures the characteristic polynomial of the corresponding binary matrix is primitive. Table 10 shows the values of these parameters. The definition and measurement of the stringency level of a test against SHR3s are identical to those of MLCGs. Figure 11 shows the stringencies of the collision tests of $m = 2^{20}$ and $n = 2^{10}, 2^{11}, \ldots, 2^{28}$. The result is similar to that of the previous two families except that the collision test suddenly loses it power against this family when too many balls are thrown, say, $n \geq 2^{24}$.



	Bit-length	Left-shift (l1)	Right-shift (r)	Left-shift (l2)
1	21	3	7	7
2	22	3	7	7
3	23	5	7	3
4	24	5	7	3
5	25	7	11	5
6	26	5	5	11
7	27	11	11	5
8	28	11	5	17
9	29	11	5	17
10	30	17	17	11
11	31	17	5	13
12	32	13	17	5
13	33	17	7	13
14	34	7	7	13
15	35	19	17	11
16	36	11	11	19
17	37	13	7	19
18	38	7	7	17
19	39	15	5	19
20	40	7	11	19
21	41	7	5	19
22	42	13	11	17
23	43	15	13	17
24	44	11	13	25
25	45	19	13	25
26	46	19	13	25
27	47	21	23	25
28	48	13	13	25

Table 10. The values of the (*l1*, *r*, *l2*)'s of the canonical LFGs.

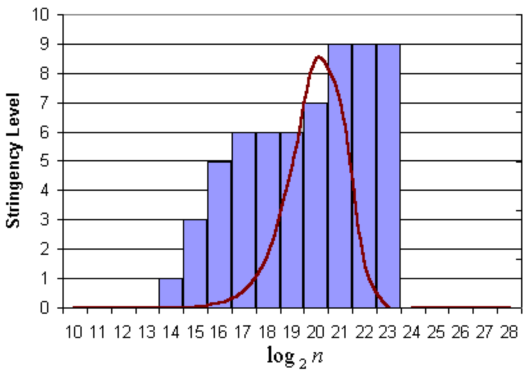
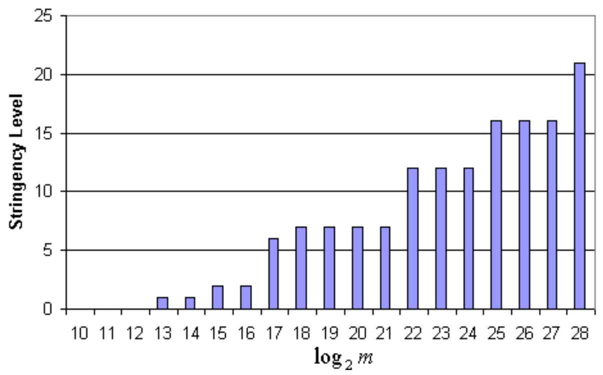


Figure 11. The stringencies against SHR3 and variance of collision tests with $m = 2^{20}$.

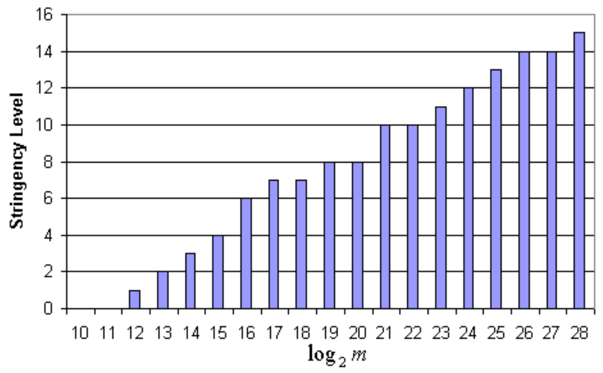
5 A scalable collision test

A C function which conducts the collision test with $n = \lfloor 1.256431m \rfloor$ was implemented (available at *http://www.csis.hku.hk/~tsang/*). The number of urns, *m*, is a scalable parameter and is restricted to powers of two. The run-time complexity of the test is O(*m*) and the RAM required is *m* bits. Figures 12, 13 and 14 shows the stringencies of the tests with $m = 2^{10}, 2^{11}, \ldots, 2^{28}$ against the MLCGs, LFGs and SHR3s, respectively. As expected, the power of the test gradually increases when *m* increases.

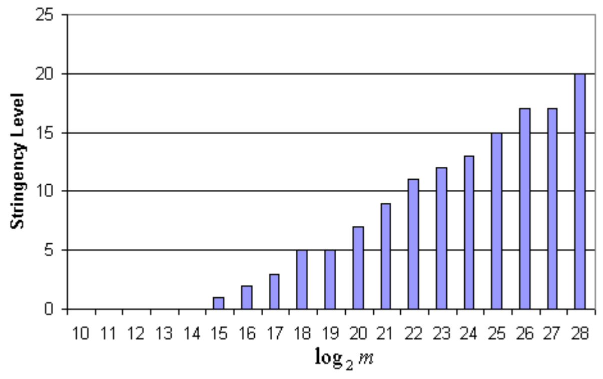
To demonstrate the power of the test, we applied the test to eighteen common RNGs. For each generator, we examined either the sequence of MSB or that of LSB. In general, the MSB sequence is at least as random as the sequences formed from bits of other positions. A failure of MSB sequence implies the failure of most of the other sequences. On the other hand, the LSB sequence is most vulnerable. All other sequences are likely to pass the test if even the LSB passes it.



Figures 12. The stringencies of the tests against the MLCGs.



Figures 13. The stringencies of the tests against the LFGs.



Figures 14. The stringencies of the tests against the SHR3s.

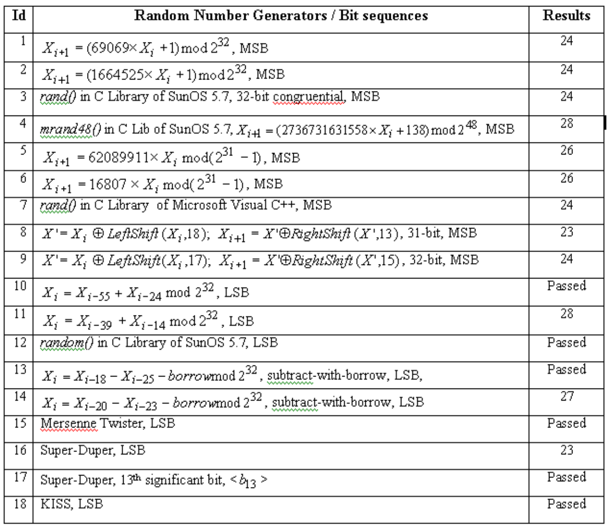
To examine a sequence, we conducted the test with $m = 2^i$, for i = 21, 22, …, up to 30. A good sequence would pass all the 20 rounds of the test and we mark a “Passed” in Table 15. A poor sequence would typically pass in the first few rounds and than kept failing in the rest. The index of the test, $i = \log_2 m$, that the generator started to fail was recorded in the last column.

The first three generators in Table 15 are congruential generators of modulus equal to 2^{32}. The multiplier of the first one was suggested by G. Marsaglia [Marsaglia 1972] and that of the second one was suggested by M. Lavaux and F. Janssens. Their behaviours against the collision tests are similar. The LSB has a period of only two and is the least random. The MSB is better blended but nonetheless was flunked by the collision tests of $m \geq 2^{24}$. The 4th one is a 48-bit generator of the same kind and its MSB failed when $m \geq 2^{28}$. The 5th and the 6th one are 31-bit congruential generators of modulus not equal to powers of 2 [Fishman 1986, Lewis 1969]. The randomness of the LSB is about the same as that of the MSB. They sustained the collision test better and only broke down when $m \geq 2^{26}$. The 7th generator is provided by Microsoft Visual C++ and its MSB failed the test of $m \geq 2^{24}$. The manual does not mention what kind of generator it is. From the execution time it takes and the changes of randomness in the bit sequences, we guess that it is a congruential generator of modulus equal to 2^{32}.

The 8th and the 9th are 31-bit and 32-bit shift-register generators [Marsaglia 1983]. The randomness of their LSB and MSB are roughly the same. Their MSBs failed when $m \geq 2^{24}$. The 10th is a lagged Fibonacci generator devised by G.J. Mitchell and D.P. Moore. That its LSB passed implies that the generator passed. The 11th is another additive generator with smaller lags. Its LSB marginally failed when $m = 2^{28}$. The lesson is: do not use any lags less than those of the 10th. The 12th is described as a nonlinear additive feedback random number generator in the manual. It passed the test.

The 13th and 14th are examples of subtract-with-borrow generators. The generators of this new class generally pass the collision test unless the lags are less than 25 or so. The 15th was suggested by Makoto Matsumoto and Takuji Nishimura [Matsumoto 1998]. It keeps 624 words and is backed with strong theory. It passed the tests as expected.

The generator tested in the 16th and 17th rows is Super-Duper in the McGill Random Number Package which was implemented by G. Marsaglia in the 70’s. It combines the outputs of the 1st and the 9th generator and it has been noted that the least significant bits were not very random [Marsaglia 1984]. We found that the LSB sequence ($<b_1>$) failed the test at $m = 2^{23}$ and so did the second LSB sequence ($<b_2>$), the third LSB sequence ($<b_3>$), …, up to $<b_{12}>$ at larger m’s. Starting from $<b_{13}>$, all the higher order bit sequences passed. The name of the KISS generator in the 18th row stands for *Keep it Simple Stupid.* It was implemented and disseminated through the Internet by G. Marsaglia in January 1999. It combines a congruential, a shift-register and a multiply-with-carry generators and made a clear-cut pass.



Id	Random Number Generators / Bit sequences	Results
1	$X_{i+1} = (69069 \times X_i + 1) \bmod 2^{32}$, MSB	24
2	$X_{i+1} = (1664525 \times X_i + 1) \bmod 2^{32}$, MSB	24
3	*rand()* in C Library of SunOS 5.7, 32-bit congruential, MSB	24
4	*mrand48()* in C Lib of SunOS 5.7, $X_{i+1} = (2736731631558 \times X_i + 138) \bmod 2^{48}$, MSB	28
5	$X_{i+1} = 62089911 \times X_i \bmod (2^{31} - 1)$, MSB	26
6	$X_{i+1} = 16807 \times X_i \bmod (2^{31} - 1)$, MSB	26
7	*rand()* in C Library of Microsoft Visual C++, MSB	24
8	$X' = X_i \oplus \mathit{LeftShift}(X_i, 18);\ X_{i+1} = X' \oplus \mathit{RightShift}(X', 13)$, 31-bit, MSB	23
9	$X' = X_i \oplus \mathit{LeftShift}(X_i, 17);\ X_{i+1} = X' \oplus \mathit{RightShift}(X', 15)$, 32-bit, MSB	24
10	$X_i = X_{i-55} + X_{i-24} \bmod 2^{32}$, LSB	Passed
11	$X_i = X_{i-39} + X_{i-14} \bmod 2^{32}$, LSB	28
12	*random()* in C Library of SunOS 5.7, LSB	Passed
13	$X_i = X_{i-18} - X_{i-25} - \mathit{borrow} \bmod 2^{32}$, subtract-with-borrow, LSB,	Passed
14	$X_i = X_{i-20} - X_{i-23} - \mathit{borrow} \bmod 2^{32}$, subtract-with-borrow, LSB	27
15	Mersenne Twister, LSB	Passed
16	Super-Duper, LSB	23
17	Super-Duper, 13th significant bit, $< b_{13} >$	Passed
18	KISS, LSB	Passed

Table 15. The testing results of many common random number generators

6 References

Feller, W., 1950, *An Introduction to Probability Theory and its Applications, Vol. 1.*, John Wiley & Sons.

Fishman, G. S., and Moore III, L. R., 1986, An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$, *SIAM J. Sci. Stat. Comput*. **7**, 24-45.

Golomb, S. W., 1982, *Shift Register Sequences, Rev. ed.*, Aegean Park Press.

Knuth, D. E., 1981, *The Art of Computer Programming, Vol. 2, 2nd ed.*, Addison-Wesley.

Knuth, D. E., 1997, *The Art of Computer Programming, Vol. 2, 3rd ed.*, Addison-Wesley.

L’Ecuyer, P. 1999, Tables of linear congruential generators of different sizes and good lattice structure, *Mathematics of Computation*, 68, 225, 249—260.

Lewis, Goodman, and Miller, 1969, A Pseudo-random Number Generator for the IBM 360, *IBM Systems J.* **8**, 136-146.

Marsaglia, G., 1972, *The structure of linear congruential sequences, Applications of Number Theory to Numerical Analysis, Z. K. Zaremba, ed.*, New York: Academic Press, 249-285.

Marsaglia, G., 1983, Random number generation, *Encyclopedia of Computer Science and Engineering, 2nd ed.*, Van Nostrand Reinhold.

Marsaglia, G., 1984, A current View of Random Number Generators, Keynote Address, *Computer Science and Statistics: 16th Symposium on the Interface*, Atlanta.

Marsaglia, G., and Tsay, Liang-Huei, 1985, Matrices and the Structure of Random Number Sequences, *Linear Algebra and Its Applications*, **67**, 147-185.

Marsaglia, G., and Zaman, A., 1991, A new class of random number generators, *The Annals of Applied Probability 1*, No. **3**, 462-480.

Marsaglia, G., and Zaman, A., 1993, Monkey tests for random number generators, *Computers and Mathematics with Applications* **26**, 9, 1-10.

Marsaglia, G., 1999, Random numbers for C: End, at last, sci.stat.math Web discussion, January 21.

Matsumoto, M., and Nishimura, T., 1998, Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Trans. Model. Comput. Simul.* **8**, No. 1, 3-30.

Vattulainen, Kankaala, K., Saarinen, J., and Ala-Nissila, T., 1995, A comparative study of some pseudorandom number generators, *Computer Physics Communications* **86**, 209-226.