Tuning the Collision Test for Stringency

W.W. Tsang, L.C.K. Hui, K.P. Chow and C.F. Chong
The University of Hong Kong
Email: *tsang@csis.hku.hk*

The collision test is one of the most important statistical tests for random number generators. It 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 stringent possible on a computer available for testing. Our studies showed that for a fixed *m*, *n* shall be determined subject to maximizing the variance of the number of collisions. With this rule, the stringency of the resulting test increases as *m* increases. And the test of $m \geq 2^{23}$ flunked a recorded number of generators, including congruential generators, shift-register generators, additive generators of lags less than 40, subtract-with-borrow generators of lags less than 24, and a combination of a congruential and a shift-register generator.

Key Words: Random number testing, Statistical tests, Collision test

1. INTRODUCTION

The collision test suggested by H. Delgas Christiansen in 1975 is among the foremost statistical tests for random number generators. The test simulates throwing balls randomly into urns. The number of urns, *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*. A random number generator fails the test if *c* falls outside a predefined interval. Let *n* be the number of ball thrown. The test requires *m* bits in RAM to keep track of the statuses of urns and the run-time complexity is O(*n*).

One reason that the collision test is important is that throwing of balls is identical to insertions of items into a hash table and collisions are a major concern in both cases. It is one of a handful of statistical tests for random number generators 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 random number generators [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 *stringent*, 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 stringency on a computer available for testing? Can the stringency 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 determined subject to maximizing *c*. The stringency of the test with *n* determined this way increases as *m* increases. This conclusion relies on a bold attempt in quantifying the stringency of a test.

First, we chose 64 sequences of bits whose randomness is increasing. The *stringency level* of a collision test is then defined to be the number of sequences that the test flunks. To verify the appropriateness of the definition, we have worked out the stringency levels of numerous collision tests of different *m* and *n* values and have applied the tests on random number generators of different kinds. The results confirm that the stringency level so obtained does indicate a test's general ability in rejecting bad generators.

With this quantitative measurement for stringency, we found that for a fixed *m*, when *n* increases, the stringency level increases but eventually levels off. As additional effort is needed but no stringency is gained when *n* increases beyond a threshold, this threshold value is a good choice for *n* in the collision test. Further investigation showed that 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 threshold. Thus, *n* shall be determined subject to maximizing the variance of *c*. With this *maximum variance criterion*, we found that asymptotically, $n = 1.256431m$.

The collision test $n = 1.25m$, with changeable *m* was implemented. As expected, the stringency of the test increases when *m* increases. A generator being examined is tried out with this test of $m = 2^{21}$, 2^{22}, …, up to 2^{30} one by one. Many well-known generators of various kinds were flunked starting from some point on 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 additive 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 complete 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 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 stringency. Starting from quantifying the stringency, we devise the maximum variance criterion, and explain how to reach the conclusion that *n* approaches 1.256431*m* asymptotically. Finally, we applied the fine tuned collision tests of various *m* on many commonly known 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 $\frac{m(m-1)\cdots(m-n+c+1)}{m^n}\left\{ \begin{matrix} n \\ n-c \end{matrix} \right\}$, where $\left\{ \begin{matrix} n \\ k \end{matrix} \right\}$ is a Sterling number of 2nd kind

defined as $\left\{ \begin{matrix} n \\ 1 \end{matrix} \right\} = 1,\ \left\{ \begin{matrix} n \\ n \end{matrix} \right\} = 1,$ otherwise $\left\{ \begin{matrix} n \\ k \end{matrix} \right\} = k\left\{ \begin{matrix} n-1 \\ k \end{matrix} \right\} + \left\{ \begin{matrix} n-1 \\ k-1 \end{matrix} \right\}$ [Knuth 1997].

Based on a recursion derived from 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 the array A[]. The latter is of order the square root of the variance of *c*. Using MAPLE, the Taylor expansion of Formula (3.1) for the variance given in next section are

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

When $n = m$, 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 450MHz PC. When $n = m \geq 2^{21}$, *pcoll1()* will take too long to complete and a more efficient method is needed.

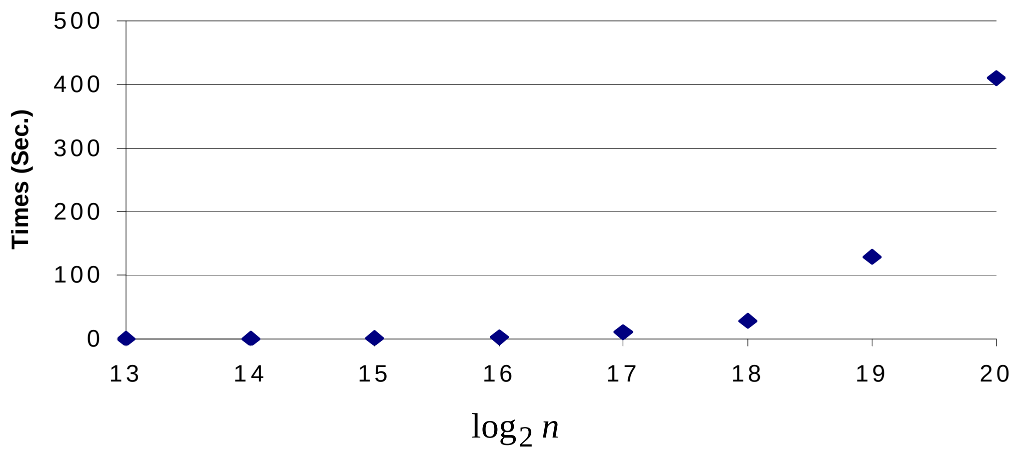


Figure 2. The execution time of *pcoll1()* versus $\log_2 n$, for $m = 2^{20}$.

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 the mean, $l = me^{-n/m}$, under the conditions that *m* and *n* are large and that l remains bounded. As a Poisson distribution approximates to normal when l increases, *e* asymptotically follows the normal distribution with both the mean and variance equal to l. 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*, s_e^2, instead of l [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

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

Since $e = X_1 + X_2 + \cdots + X_m$, $m_e = mE(X_1) = mq$. Furthermore,

$$s_e^2 = Var(X_1) + \cdots + Var(X_m) + \sum_i \sum_{j \neq i} Cov(X_i, X_j)$$

$$= mVar(X_1) + (m^2 - m)Cov(X_1, X_2)$$

$$= m(q - q^2) + (m^2 - m)(r - q^2)$$

$$= m(q + mr - r - mq^2)$$

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

$$m_c = m_e - m + n = mq - m + n, \text{ and}$$

$$s_c^2 = s_e^2 = m(q + mr - r - mq^2). \qquad (3.1)$$

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.

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

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

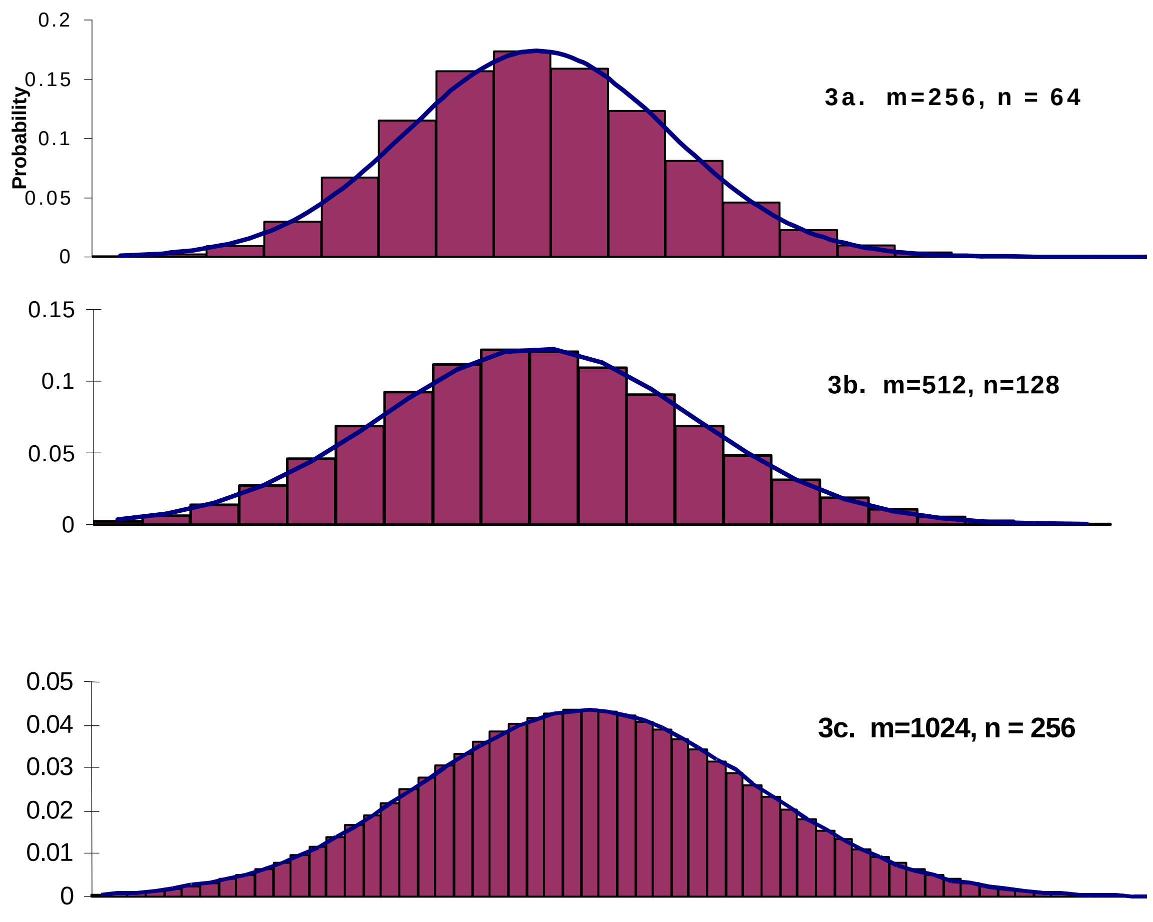


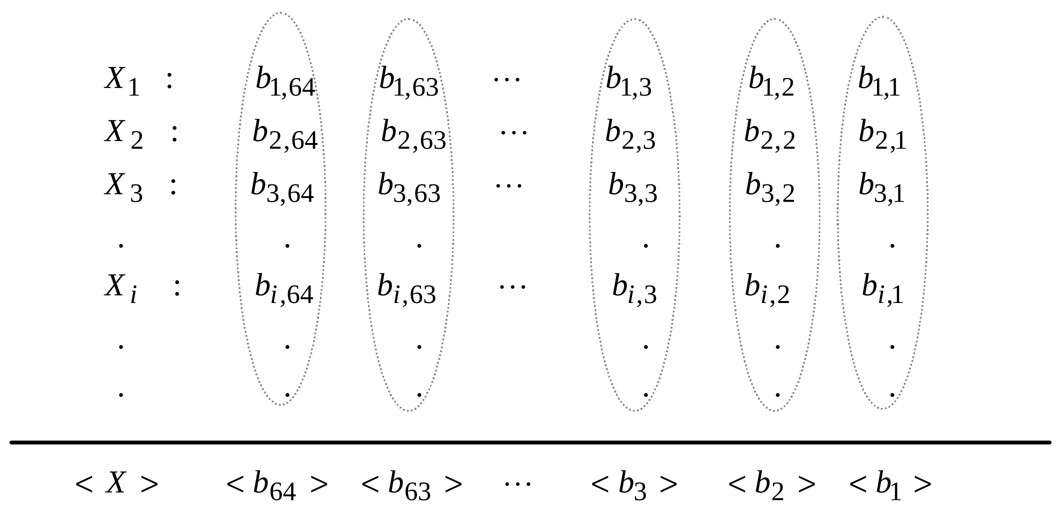
Figure 3. The exact distributions of c and their normal approximations.

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 more stringent test. So, what will be a good choice for *n*? As the collision test is used to examine random number generators, we will like to choose *n* such that the test has the highest ability in rejecting bad generators. This problem was tackled empirically. First, we choose 64 bit sequences whose randomness is believed to be increasing. To gauge the stringency of a particular collision test, we conduct the test on the sequences one by one, from the least random to the most random. The stringent level of the test is defined as the number of sequences it flunks, before the first sequence that it passes. With this quantitative measurement for the stringency, we find *n* which leads to the highest stringency level.

Consider the 64-bit congruential generator due to C.E. Haynes shown in Figure 5. The bits of X_1 is $b_{1,64}b_{1,63}\cdots b_{1,1}$, where $b_{1,1}$ is the least significant bit. The bits of X_2 is $b_{2,64}b_{2,63}\cdots b_{2,1}$, and so on so forth. The bit sequence $<b_1>$ consists of the least significant bits of $X_1, X_2, \cdots$. In general, $<b_k>$ consists of the k^{th} least significant bit sequences of $X_1, X_2, \cdots$. It has been known that the least significant bits of congruential generators with moduli equal to powers of 2 are not as random as the most significant bits [Marsaglia 1984]. After all, the period of $<b_k>$ is bounded by 2^k because the least *k* significant bits of X_i only depend on the least *k* significant bits of X_{i-1}. It makes good sense to anticipate that $<b_k>$ is less random than $<b_{k+1}>$, for $k = 1$ to 63. Now, if a collision test rejects the sequences $<b_1>, <b_2>, \ldots, <b_k>$ but not $<b_{k+1}>$, we say that its stringency level is *k*.

$$X_{i+1} = 6364136223\ 846793005 \times X_i + 428856369 \mod 2^{64}$$



X_1 : $b_{1,64}$ $b_{1,63}$ $\cdots$ $b_{1,3}$ $b_{1,2}$ $b_{1,1}$
X_2 : $b_{2,64}$ $b_{2,63}$ $\cdots$ $b_{2,3}$ $b_{2,2}$ $b_{2,1}$
X_3 : $b_{3,64}$ $b_{3,63}$ $\cdots$ $b_{3,3}$ $b_{3,2}$ $b_{3,1}$
X_i : $b_{i,64}$ $b_{i,63}$ $\cdots$ $b_{i,3}$ $b_{i,2}$ $b_{i,1}$
$<X>$ $<b_{64}>$ $<b_{63}>$ $\cdots$ $<b_3>$ $<b_2>$ $<b_1>$

Figure 5. The bit sequences extracted from a 64-bit congruential generator.

We had applied the collision test of $m = 2^{20}$ and $n = 2^{14}$ to the $<b_k>$'s. The resulting number of collisions obtained in testing each bit sequence was converted to a uniform random number, *U*, using *pcoll1()*. The test flunks a sequence when $U < 0.001$ or $U > 0.999$. The outputs are shown in Figure 6. According to our definition, the stringency level of the collision test is 15.

```
U:<b 1>...<b 8>| 1.0000 1.0000 1.0000  1.0000  1.0000  1.0000  1.0000  1.0000
U:<b 9>...<b16>| 1.0000 1.0000 1.0000  1.0000  1.0000  1.0000  1.0000  0.0432
U:<b17>...<b24>| 0.0759 0.7699 0.9111  0.1663  0.8415  0.9956  0.1663  0.1902
U:<b25>...<b32>| 0.4052 0.9956 0.0352  0.6168  0.1663  0.7956  0.6824  0.5825
U:<b33>...<b40>| 0.3045 0.7424 0.8195  0.2439  0.1902  0.6502  0.6502  0.9111
U:<b41>...<b48>| 0.7424 0.6168 0.7131  0.4761  0.6502  0.0902  0.0432  0.4052
U:<b49>...<b56>| 0.0526 0.6168 0.5474  0.2161  0.8415  0.8616  0.1443  0.5474
U:<b57>...<b64>| 0.7956 0.8799 0.5474  0.5825  0.7699  0.5118  0.1663  0.5474
The discriminating power is 15
```

Figure 6. The outcome of applying the collision test of $m = 2^{20}$ and $n = 2^{14}$ to $<b_k>$'s.

As a typical example, the stringency levels of the collision tests of $m = 2^{20}$ against various *n* values are shown as bars in Figure 7. The stringency increases as *n* increases until $n = 2^{20}$. Thereafter, the stringency remains more or less constant. To understand such behavior, we superimposed the curve of the variances of *c* corresponding to the collision tests in the bar chart (with different scale in vertical axis). 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

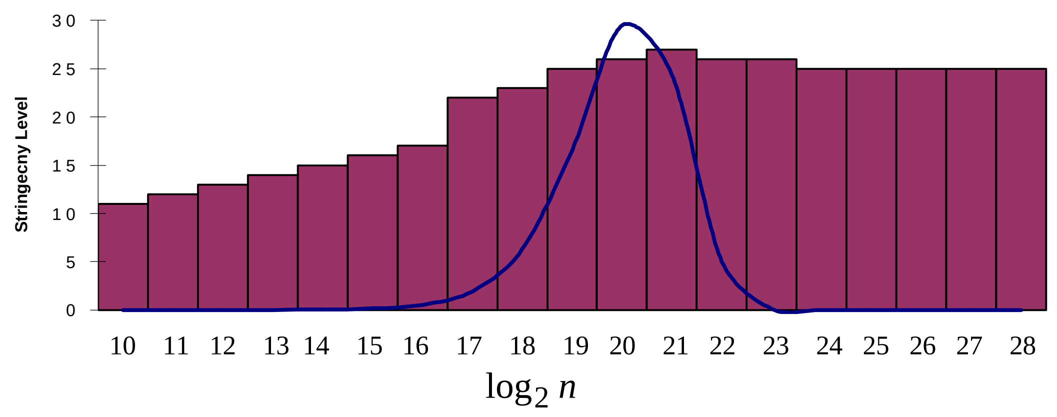


Figure 7. The discriminating powers and variances of monkey tests of $m = 2^{20}$.

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 flunking 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 the high level and forms a plateau. It is due to the fact that some urns remain empty no matter how many balls have been thrown.

From the above empirical results, we should choose $n \cong n_v$ since additional effort is needed but no stringency is gained when *n* increases beyond n_v. 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. Thus, we suggest to choose *n* equal to $\lfloor 1.256431m \rfloor$ in the collision test for $m \geq 2^{20}$.

Figure 8. shown the stringency levels of collision tests of $n = \lfloor 1.256431m \rfloor$ against *m*. As expected, the stringency generally increases when *m* increases. The run-time complexity of the test is O(*m*) and the RAM required is *m* bits.

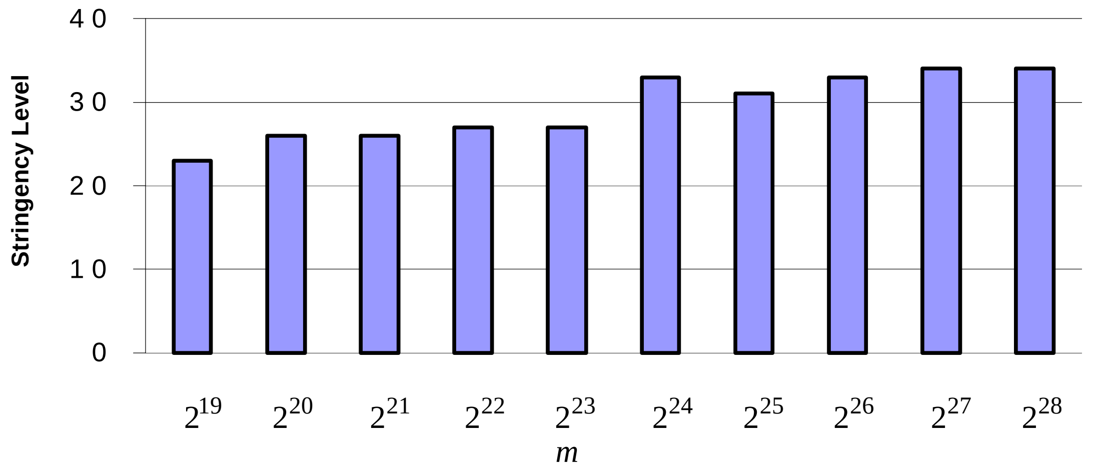


Figure 8. The discriminating power of various collision tests of $n = 1.256431m$.

5. THE DISCRIMINATING POWER

A C function which conducts the collision test of $n = 1.256431m$ was implemented (available at *http://www.csis.hku.hk/~tsang/*). The number of urns, m, is a changeable parameter and is restricted to powers of two. To test a 32-bit random number generator, we first decide which bit sequence, $<b_k>$, defined in Section 4, that is going to be examined. In general, the most significant bit sequence (MSB), $<b_{32}>$, is at least as random as the other bit sequences. A failure of MSB implies the failure of most of the other sequences. On the other hand, the least significant bit sequence (LSB), $<b_1>$, is most vulnerable. All other sequences are likely to pass the test if even the LSB passes it. For a given sequence, we conduct the test with $m = 2^i$, for i = 21, 22, …, up to 30. A sufficiently random sequence will pass all the 20 rounds of the test and we will mark a "Passed" in Table 1. A sequence with deficiencies will typically pass for the first few rounds and than keep failing for the rest. The index of the test, $i = \log_2 m$, that the generator starts to fail is recorded.

The first three generators in Table 1 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 behaviors against the collision tests are similar. The LSB has a period of only two and is 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 an additive 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.

Id	Random Number Generators / Bit sequences	Outcome
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 1. The testing results of many common random number generators

The generator tested in the16th 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 the1st 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 failed the test at $m = 2^{23}$ and so did $<b_2>$, $<b_3>$, …, 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 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.

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

Lewis, Goodman, and Miller, 1969, ?????, *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 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.