Journal of the Operations Research
Society of Japan
Vol. 41, No. 3, September 1998

© 1998 The Operations Research Society of Japan

ANALYSIS OF THE ANOMALY OF ran1() GENERATOR IN MONTE CARLO PRICING OF FINANCIAL DERIVATIVES

Akira Tajima   Syoiti Ninomiya   Shu Tezuka
IBM Research, Tokyo Research Laboratory

(Received March 6, 1997; Revised July 25, 1997)

Abstract Recently, Paskov reported that the use of a certain pseudo-random number generator, ran1(), given in Numerical Recipes in C, First Edition makes Monte Carlo simulations for pricing financial derivatives converge to wrong values.

In this paper, we trace Paskov's experiment, investigate the characteristics and the generation algorithm of the pseudo-random number generator in question, and explain why the wrong convergences occur. We then present a method for avoiding such wrong convergences. A variance reduction procedure is applied, together with a method for obtaining more precise values, and its correctness is examined. We also investigate whether statistical tests for pseudo-random numbers can detect the cause of wrong convergences.

1. Introduction

Monte Carlo methods are effective for pricing some financial derivatives, especially when the price cannot be calculated analytically because it depends on the historical movement of the underlying variable or on multiple underlying variables [3]. For example, the payoffs of look-back options depend on the maximum or minimum prices of underlying assets during the lives of the options. The remaining annuities of Mortgage-Backed Securities depend on the interest rates of the past, as we will describe in Section 2. The prices of such options are therefore difficult to calculate analytically, but easy to estimate by using Monte Carlo simulation.

However, it is generally known that pseudo-random numbers can produce pathological results in Monte Carlo simulations. Further, this phenomenon is not restricted to poor or simple random number generators. In the Physics case reported by Ferrenberg [2], the use of a high-quality generator gave incorrect results, while a simple linear congruential generator gave correct results. In financial applications, Paskov reported that the use of a certain pseudo-random number generator, ran1(), which is given in Numerical Recipes in C, First Edition [9], makes Monte Carlo pricing of derivatives converge to wrong values [8]. We will focus on this phenomenon in the present paper.

In recent years, low-discrepancy sequences have been investigated as substitutes for pseudo-random numbers in financial applications. Paskov investigated the effectiveness of low-discrepancy sequences in pricing a Collateralized Mortgage Obligation (CMO) [8]. Morokoff pointed out that in high dimensions, low-discrepancy sequences are no more uniform than pseudo random numbers [5]. Although it is known that low-discrepancy sequences improve the convergence speed, their applications are still restricted.

Consequently, there is no almighty sequence, and the Monte Carlo method with "pseudo-random" numbers is still effective. Further, it is important to know the cause of pathological results if they occur.

The objectives of this paper are to identify why the simulation converged to wrong values with ran1() in Paskov's experiment, and to investigate the anomaly so that it can be avoided in future. In section 2, we trace Paskov's experiment by using a pass-through Mortgage Backed Security (MBS), which is simpler than the CMO used by Paskov in the sense that it has no tranches. In section 3, we introduce the algorithm of the ran1() generator. In section 4, we analyze ran1() and show why Monte Carlo simulations converge to wrong values. We also present a method for avoiding the anomaly, and apply a variance reduction procedure to determine whether the results are precisely correct. In section 5, we investigate whether a statistical test for random numbers can detect the anomaly. Finally, we summarize our findings in section 6. This paper is a detailed and expanded version of Tajima, Ninomiya, and Tezuka [11].

2. Pricing Mortgage-Backed Securities (MBS)

In this section, we trace Paskov's experiment [8], in which Monte Carlo simulations for estimating the present value of Collateralized Mortgage Obligation (CMO) converged to wrong values when the random number generator ran1() [9] was used.

2.1. MBS specification

Paskov does not give the specification of the tranches of the CMO, so we simplify the problem into a pass-through MBS with no tranches. As we will show in Section 2.2, our simplification is sufficient for the same anomaly to be observed. The underlying pool of mortgages has a maturity of thirty years, and a cash flow occurs every month; in other words, there are 360 cash flows, and the dimension of the problem is also 360. Let C be the monthly payment on the underlying pool of mortgages.

For $1 \leq k \leq 360$,

r_k – the appropriate interest rate;
w_k – the percentage of pre-payment;
$a_{360-k+1}$ – the remaining annuity;
u_k – the discount factor.

Then, a_k is given by

$$a_k = \sum_{i=0}^{n}(1 + \frac{1}{1+r_0}), \tag{2.1}$$

where r_0 is the current monthly interest rate.

C and a_k are assumed to be constants. r_k, w_k, and u_k are stochastic variables, defined below. The interest rate follows the lognormal model, and the variable r_k is computed as

$$r_k = K_0 e^{\sigma \xi_k} r_{k-1} = K_0^k e^{\sigma \sum_{i=1}^{k} \xi_i} r_0, \tag{2.2}$$

where $\xi_k (1 \leq k \leq 360)$ is an independent normal random variable $N(0,1)$. Then, the discount factor u_k is given by

$$u_k = \prod_{i=0}^{k-1}(\frac{1}{1+r_i}) \tag{2.3}$$

The prepayment model w_k is a function of r_k, and is computed as

$$w_k = K_1 + K_2 \arctan(K_3 r_k + K_4) \tag{2.4}$$

The cash flow in month k is

$$M_k = C(1-w_1)\cdots(1-w_{k-1})(1-w_k+w_k a_{360-k+1}). \tag{2.5}$$

Thus, the present value of the security is the sum of the present values of the cash flows in all the months,

$$PV = \sum_{k=1}^{360} M_k u_k(\xi_1, \ldots, \xi_{360}). \tag{2.6}$$

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

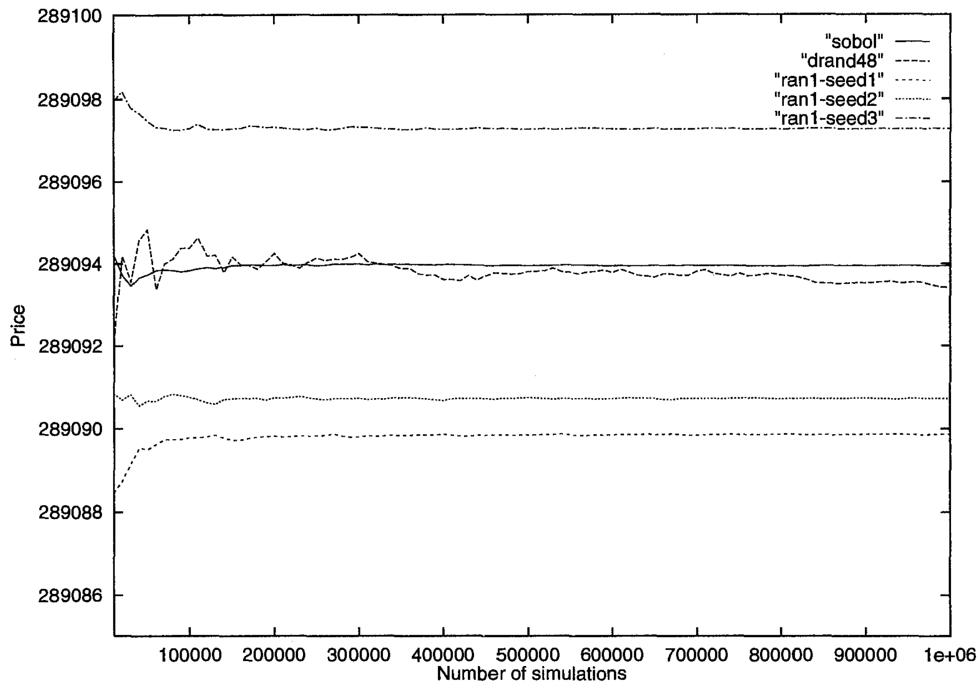

Figure 1: Monte Carlo Simulations Using Sobol and ran1() with Three Seeds

In this experiment, we set the constants as follows:

$K_0 = 1/1.020201$	$r_0 = 0.075/12$
$K_1 = 0.24$	$\sigma = 0.02$
$K_2 = 0.134$	$C = 2000$
$K_3 = -261.17 \times 12$	$a_0 = 0$
$K_4 = 12.72$	$w_0 = 0$

To generate normal random variables $N(0,1)$ from uniform random numbers [0,1), we used the inverse method given in RISK [14]. Obviously, 360 numbers are necessary for one path of Monte Carlo simulation. When we use pseudo-random generators such as ran1(), we assign the numbers from one generator to 360 months sequentially. On the other hand, in the case with low-discrepancy sequences, we assign the i-th dimension of the sequence to the i-th month $(1 \leq i \leq 360)$. Consequently, if we focus on the correlation problem among 360 variables, the serial correlations matter most with pseudo-random generators, whereas the independence among dimensions matters most with low-discrepancy sequences.

2.2. Pricing using ran1()

Figure 1 shows the result of Monte Carlo simulations. The simulation using Sobol sequences converges to the correct value. But simulations using ran1() converge to three different values, all incorrect. This result is consistent with Paskov's study. For comparison, a result using drand48(), a system-supplied random number generator of IBM xlC, is also presented. For details of Sobol sequences, see Sobol [10].

3. ran1() in Numerical Recipes in C, First Edition

In this section, we investigate the characteristics of ran1() in Press et al. [9].

The ran1() algorithm is based on three linear congruential generators. Linear congruential generators generate a sequence of integers between 0 and $m-1$ by means of the

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

recurrence relation

$$I_{j+1} = aI_j + c \pmod{m}. \tag{3.1}$$

If m, a, and c are properly chosen, the period of the sequence will be equal to m.

In ran1(), one generator ix_1 with $m_1 = 259200$, $a_1 = 7141$, and $c_1 = 54773$, and another generator ix_2 with $m_2 = 134456$, $a_2 = 8121$, and $c_2 = 28411$, are used to generate a fractional value $x[0,1)$ as

$$x = (ix_1 + ix_2 \times \frac{1}{m_2}) \times \frac{1}{m_1}. \tag{3.2}$$

The second generator, ix_3, with $m_3 = 243000$, $a_3 = 4561$, and $c_3 = 51349$, is for shuffling using 97 buckets. Shuffling is a popular method of avoiding serial correlations of the output, which have a negative effect on the results of the Monte Carlo method.

The ran1() algorithm is as follows:

Initialization: Fill the 97 buckets with fractional values.

Per Call:

1. Select a bucket by using the value generated by ix_3
2. Return the value in the bucket as the output
3. Fill the bucket with the fractional value generated by ix_1 and ix_2

4. Analysis of the Anomaly

In this section, we analyze the characteristics of ran1() and identify why the simulation converged to wrong values, then try to avoid it.

4.1. Discrepancy of points generated by ran1()

We first measure the discrepancy of the points generated by ran1(). The discrepancy can be viewed as a quantitative measure of the deviation from uniform distribution, and is related to the error of the numerical integration [7]. For N points $\mathbf{v}_0, \mathbf{v}_1, \ldots, \mathbf{v}_{N-1}$ in $[0,1]^k$, $k \geq 1$, and a subinterval $\mathbf{u} = [0, u_1) \times \cdots \times [0, u_k)$, where $0 < u_h \leq 1$ for $1 \leq h \leq k$, we define the L2-discrepancy as

$$T_N^{(k)} = (\int_{[0,1]^k} (\frac{A(\mathbf{u};N)}{N} - V(\mathbf{u}))^2 d\mathbf{u})^{1/2}, \tag{4.1}$$

where $A(\mathbf{u};N)$ is the number of values of $n, 0 \leq n < N$, with $\mathbf{v}_n \in \mathbf{u}$, and $V(\mathbf{u})$ is the volume of $\mathbf{u}$. The L2-discrepancy can be calculated in $O(N^2)$ time.

Let $\mathbf{x}_n$ be a 360-dimensional point whose i-th component is used for the i-th dimension of the n-th path of the simulation. We assign values as

$$\mathbf{x}_n = (a_{360(n-1)+1}, a_{360(n-1)+2}, \ldots, a_{360(n-1)+360}), \tag{4.2}$$

where a_i is the i-th output of ran1().

We calculate the L2-discrepancy of $\mathbf{x}_n$ (Figure 2). If the points are really random, the estimated value of L2-discrepancy is known to be

$$E((T_N^{(k)})^2) = \sqrt{\frac{2^{-k} - 3^{-k}}{N}}, \tag{4.3}$$

and it decreases as the number of points increases, as shown by the straight line in Figure 2. However, the value never falls below 2^{-230}, which is reached after 4,000 points. The movement of the value has a pattern with a cycle of about 10,000 or 11,000 points. Consequently, $\mathbf{x}_n$ has a cycle, and the accuracy of the integration does not improve after one cycle.

4.2. Cycle of ran1()

4.2.1. Floyd's Algorithm

We have observed that the points generated from the output of ran1() seems to contain a cycle. In this section, we examine whether ran1() itself contains a short cycle.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

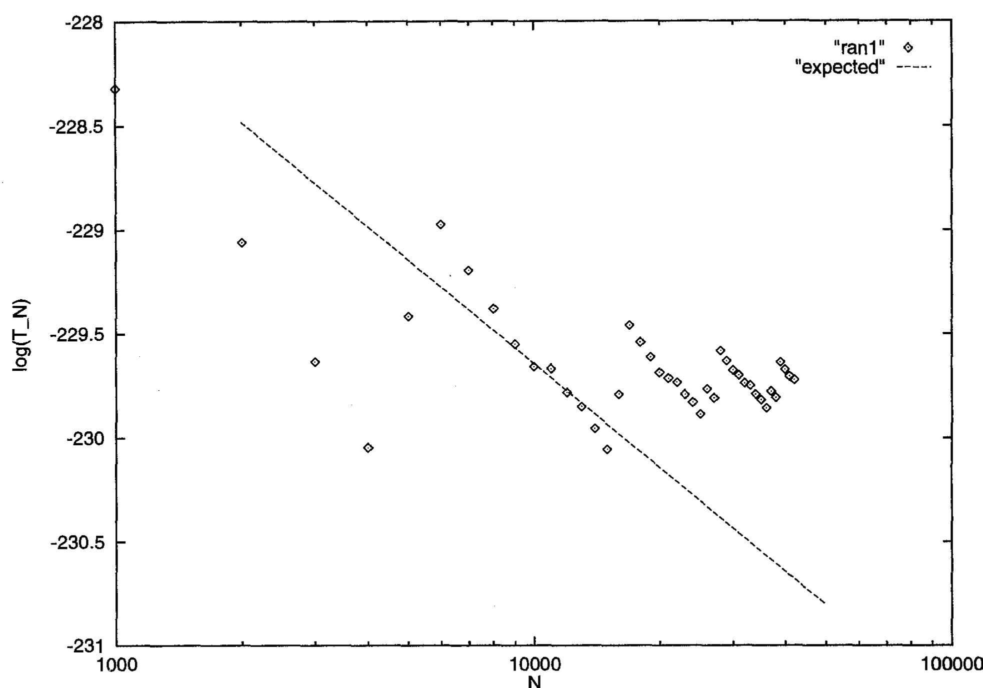

Figure 2: The L2-Discrepancy of the Points Generated by ran1()

To obtain the cycle, we apply Floyd's algorithm, given in Knuth [4]. Suppose that we have a sequence of integers $X_0, X_1, X_2, \ldots$ with the range $0 \leq X_n < m$, formed by a certain rule $X_{n+1} = f(X_n)$. Then it is known that X_n is periodic, in the sense that there exist numbers λ and μ for which although the values $X_0, X_1, \ldots, X_\mu, \ldots, X_{\mu+\lambda-1}$ are distinct, $X_{n+\lambda} = X_n$ when $n \geq \mu$. That is to say, μ is the length of the transition state and λ is the period of the stationary state.

Floyd's algorithm utilize the fact that there exists a value $n > 0$ such that $X_n = X_{2n}$ and the smallest such value of n lies in the range $\mu \leq n \leq \mu + \lambda$. Once n is found, μ is the smallest i that satisfies $X_i = X_{n+i}$. If none of the values of X_{n+i} for $0 \leq i \leq \mu$ is equal to X_n, $\lambda = n$; otherwise, λ is the smallest such i.

Using the algorithm, we examine the cycle of the sequence generated by ran1(). The algorithm can only be applied to sequences formed by the rule $X_{n+1} = f(X_n)$, as mentioned above. But ran1() does the shuffling by using multiple buckets, and the output of ran1() does not correspond to the condition in the sense that it uses a series of historical values to generate another output, as well as the last output. Here we consider a vector $\mathbf{S}_n$ that consists of all the state variables of ran1(), including ix_1, ix_2, ix_3, and the contents of the 97 buckets, instead of the output of ran1(). Thus we can say that $\mathbf{S}_{n+1} = f(\mathbf{S}_n)$ holds, and can apply the algorithm.

Unfortunately, the cycle is too large, more than 10^{10}, and we could not obtain it by computer simulations. Anyway, it is very large in comparison with the number necessary for Monte Carlo simulation of MBS, 360×10^6, and does not seem to affect the result of Monte Carlo simulations.

We therefore focus on the most significant part of the output of ran1(); that is, we modify ran1() to keep ix_2 equal to zero, and apply the same algorithm. This time we obtain the cycle given in Table 1.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Table 1: μ and λ of the Most Significant Part of ran1()

	seed	μ	λ
#1	-386834056	490	3888000
#2	-1947093824	477	3888000
#3	-1505612680	735	3888000
#4	-965150217	444	3888000
#5	-1907586699	527	3888000
#6	-1548523014	650	3888000
#7	-829591941	472	3888000
#8	-1967060894	413	3888000
#9	-1997660300	367	3888000
#10	-158225988	549	3888000
average		512.4	3888000

4.2.2. Validation of the result

In the stationary state, the cycle of the sequence λ should be equal to the least common multiple of m_1 and m_3. As in Section 3, $m_1 = 259200 = 2^7 \times 3^4 \times 5^2$, and $m_3 = 243000 = 2^3 \times 3^5 \times 5^3$, and therefore $\lambda = 2^7 \times 3^5 \times 5^3 = 3888000$. This coincides with the result in Table 1.

We then estimate the value of μ, the length of the transition state. From the generation algorithm in Section 3, the transition state is caused by the initialization of the 97 buckets. We can see that if all the 97 buckets are visited and the values in them are flushed, the effect of the initialization disappears and the generator goes into the stationary state.

If we assume that the buckets in ran1() are visited at random instead of through the value of ix_3, the expected number necessary to visit all the buckets, which is equal to the expected μ, can be calculated as in the coupon collector's problem [6],

$$E[X] = n\sum_{i=1}^{n}\frac{1}{i} = 97 \times \sum_{i=1}^{97}\frac{1}{i} = 500.23. \qquad (4.4)$$

This theoretical value is consistent with the experimental results given in Table 1.

Anyway, μ is much smaller than λ. Further, one path requires 360 random numbers, and it takes only a few paths to reach the stationary state. Hence the effect of the initial instability is subtle, and can be ignored when examining the wrong convergence.

4.3. The cycle when applied to MBSs

We continue to examine the cycle of ran1() with the precision $1/m_1$; that is, we ignore the least significant part of the value, and consider the effect of the cycle when it is applied to the pricing of MBSs.

Here, $\lambda = 3888000 = 360 \times 10800$; in other words, the cycle is a multiple of the dimension of the problem. Thus, when applied to the pricing of an MBS with 360 dimensions, there exists a cycle of $10,800$ for each dimension. If we consider the 360-dimensional point $\mathbf{x}_n$ given in equation (4.2),

$$|\mathbf{x}_{n+10800} - \mathbf{x}_n| < \frac{2}{m_1} \qquad |\mathbf{x}| : L_\infty - norm\ of\ \mathbf{x} \qquad (4.5)$$

holds for $n > \mu/360$. This inequality means that, after 10800 points, every new point falls in the neighborhood of one of the existing points. In other words, a Monte Carlo simulation for an MBS has only $10,800$ paths. This coincides with the result obtained in Section 4.1.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

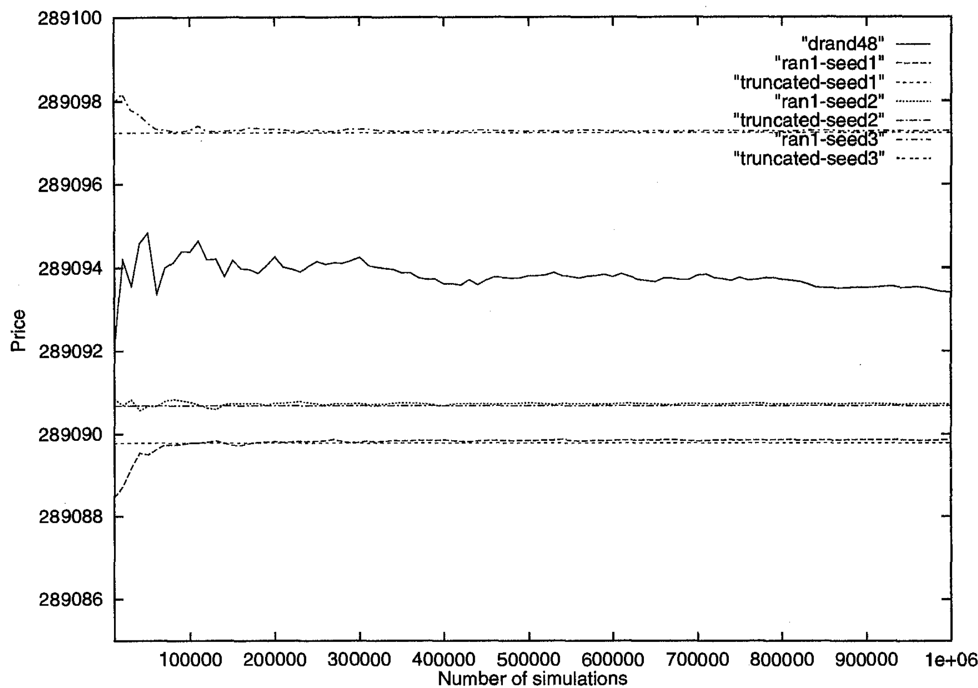

Figure 3: Simulation Almost Converges to the Value of the First 10800 Trials

To confirm that the cycle of 10800 is the cause of the wrong convergence, we compared two Monte Carlo simulations, one a simulation of 1000000 paths using ran1(), and the other the resulting value after simulation of 10800 paths using the simplified version of ran1() in which

$$x = (ix_1 + 0.5) \times \frac{1}{m_1} \tag{4.6}$$

instead of equation (3.2). That is to say, we approximate the least significant part by $0.5 \times 1/m_1$ and stopped the simulation after just one cycle.

Figure 3 shows the results with three different seeds. The full simulation converges to the value obtained with the first 10, 800 paths which is represented as a horizontal line. The amplitude is caused only by the least significant part.

4.4. Avoiding the cycle

Here we examine whether we can avoid the incorrect convergence. We have observed that the inaccuracy is caused because the cycle of ran1() is a multiple of the dimension of the problem, 360.

Considering that the cycle of the most significant part of ran1() $\lambda = 3888000 = 2^7 \times 3^5 \times 5^3$, Monte Carlo simulations will converge to the correct value if the dimension of the problem is a prime. A simple way to realize this is to skip several random numbers after every path. Namely, for the 360 dimensional point $\mathbf{x}_n$, we assign values as follows:

$$\mathbf{x}_n = \left(a_{(360+\alpha)(n-1)+1}, a_{(360+\alpha)(n-1)+2}, \cdots, a_{(360+\alpha)(n-1)+360}\right) \tag{4.7}$$

where α is the number of skipped random numbers. Figure 4 shows the result of Monte Carlo simulations using ran1() with the skip numbers 13, 41, and 143, which give the primes 373, 401, and 503 when added to 360. If we compare the results with those obtained by using drand48(), we can say that the anomaly of ran1() is removed.

Further, to check whether they converge to the correct value precisely, we also apply a variance reduction procedure, the antithetic variable technique [3]. That is to say, after

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

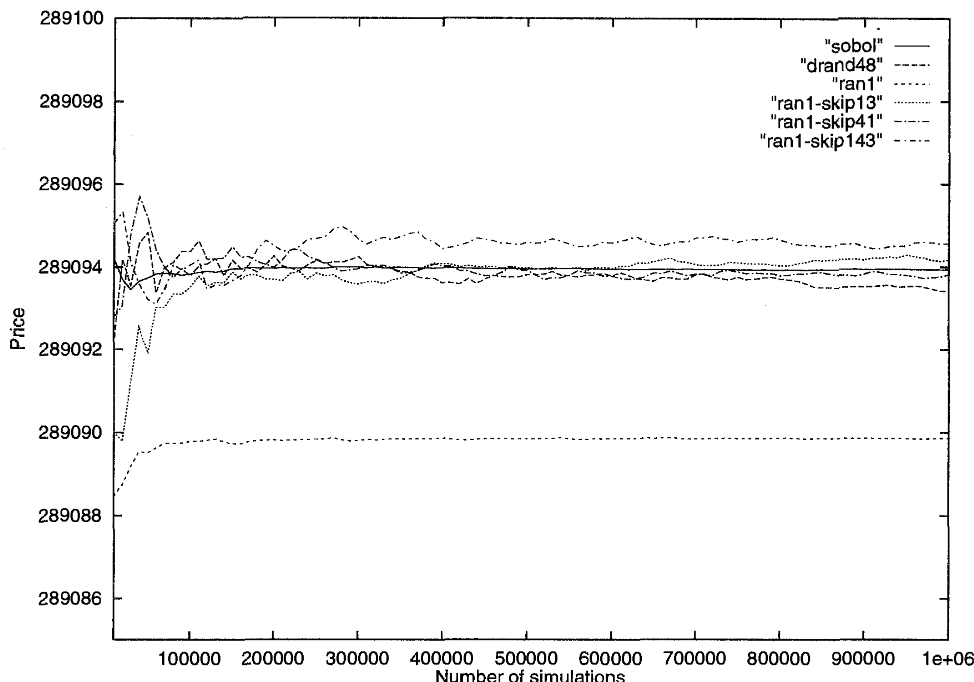

Figure 4: ran1() with Skips after Every 360 Random Numbers

calculating one path by using a normal random number vector $\mathbf{f}_n$, which is generated from the uniform random number vector $\mathbf{x}_n$ as in Section 2.1, we also calculate another path by using $-\mathbf{f}_n$. Figure 5 shows the result obtained by using this antithetic variable technique. In the figure, the X-axis represents the number of paths, not the number of random number vectors. That is to say, the number of random number vectors generated is the half of the number of paths, and is equal to 500000.

From these experiments, we conclude that if we apply some appropriate techniques, the correct value can be calculated with a good convergence speed by using ran1().

5. Statistical Tests

In this section, we investigate whether a statistical test for random sequences can detect the anomaly that we have isolated in ran1(). There are many statistical tests for checking various aspects of randomness. For various examples of these tests, see Knuth [4] or Tezuka [12]. Here, to detect the anomaly in question, the test needs to use a large number of random numbers, namely, more than 3888000. We focus on the collision test as a representative of such tests.

5.1. Collision test

The collision test is related to hashing. Consider a situation in which n balls are thrown into m buckets at random, where m is much larger than n. A collision occurs when more than one ball falls into one bucket. We can calculate the average number of collisions and the probability distribution of the number of collisions, for a given m and n. In Knuth's example [4],

$$m = 2^{20},\ n = 2^{14},\ E[\#of collisions] \simeq 128$$

$collisions \leq$	101	108	119	126	134	145	153
with probability	.009	.043	.244	.476	.742	.946	.989

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

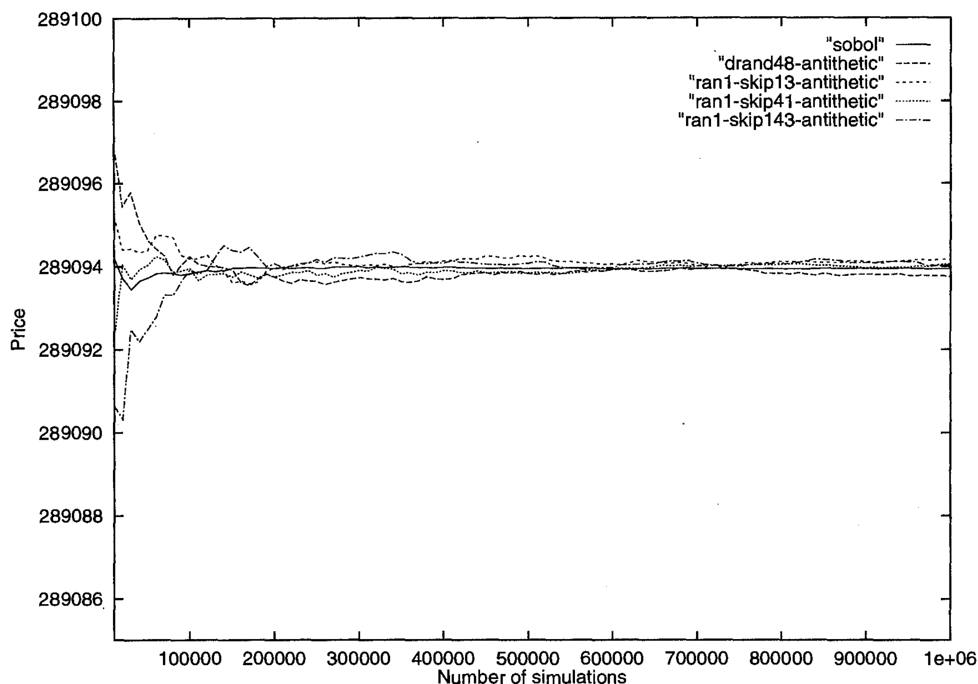

Figure 5: ran1() with Skips and Antithetic Variable Technique

By repeating this test many times independently, we can check whether the results are biased.

5.2. Looking for the settings

When the number of buckets m is equal to 2^a, we use a random numbers to select the destination of one ball. Namely, by checking whether each number is larger than 0.5 or not, we generate a-bit information. Thus, to detect the anomaly, the following conditions should be satisfied:

- $a \times n > 3888000$
- $3888000 \equiv 0 \pmod{a}$

If the test is applied to ran1() with these conditions satisfied, *every throw of a ball results in a collision*, after 3888000 random numbers have been used.

We looked for the parameters that satisfy the conditions above, as in Table 2. With the first and second settings, the estimated number of collisions is too large compared with Knuth's example in Section 5.1. If we use a smaller value of n, the collision test cannot find the anomaly. On the other hand, the third settings requires 128 megabytes of memory, and a good computing environment.

It is thus very hard to detect the anomaly by applying the collision test without knowing the existence and the characteristics of the anomaly.

5.3. Experiments

We implemented the test, with the results shown in Figure 6. Following a different approach from that described in Section 5.1, we do not fix the value of n and plot the number of collisions as the number of balls increases gradually. The results shown are those of a one-time run, and not a summary of multiple runs. But they are sufficient for us to observe the anomaly. The X-axis represents the number of outputs of ran1(), not the number of balls, so that the anomaly takes place at the same place on the X-axis.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Table 2: Possible settings of m and n

m	Memory space [MB]	n	E[# of collisions]
2^{25}	4	2^{18}	$2^{10} = 1024$
2^{27}	16	2^{18}	$2^{8} = 256$
2^{30}	128	2^{18}	$2^{5} = 32$

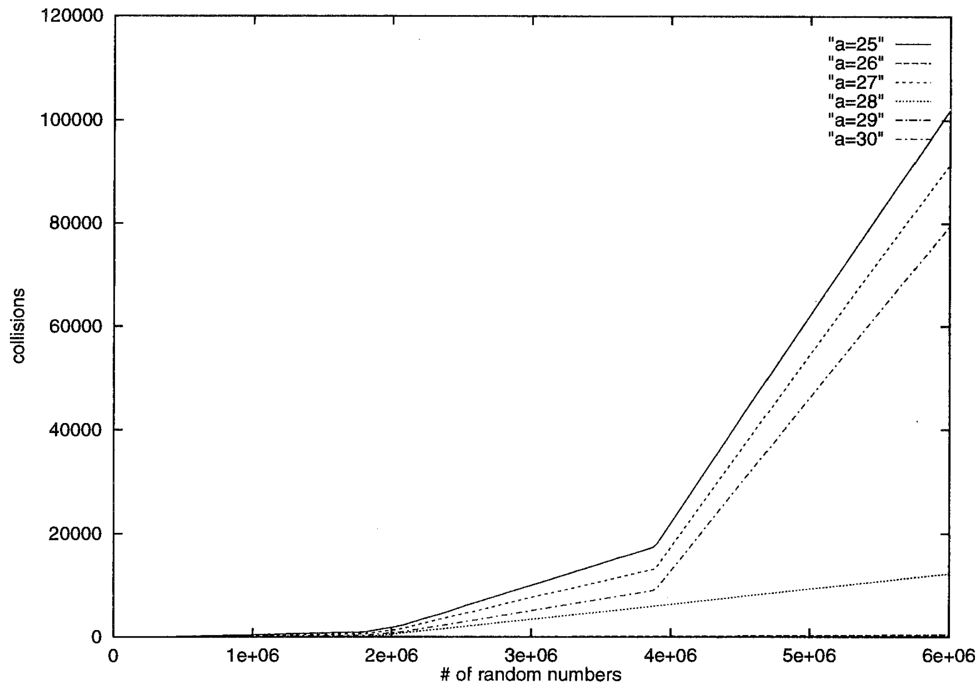

Figure 6: Number of collisions with various numbers of dimensions

The following numbers of buckets are examined: $m = 2^{25}$, 2^{26}, 2^{27}, 2^{28}, 2^{29}, and 2^{30}. Consistent with the condition in Section 5.2, the number of collisions rapidly increases after 3888000 random numbers have been used with $m = 2^{25}$, 2^{27}, and 2^{30}. But with $m = 2^{25}$,2^{27}, 2^{28}, and 2^{30}, we can see another anomaly: namely, the number of collisions is still larger than the expected value before 3888000 random numbers have been used. $m = 2^{26}$ and 2^{29} are the only settings that produce correct results.

The second anomaly can be detected more easily than the first one, and this may be a sufficient reason to avoid the use of ran1(). However, we have not isolated the cause of the second anomaly. That is to say, the number of dimensions for MBS evaluation that induces only the second anomaly, which corresponds to $m = 2^{28}$ in Figure 6, is not known. Therefore, the effect of the second anomaly on actual Monte Carlo simulations cannot be measured yet.

6. Conclusion

We have investigated why the random number generator ran1() in Press et al. [9] caused wrong convergences when used for pricing financial derivatives.

The main reason is that the cycle of ran1() is a multiple of the number of dimensions of the problem, 360. Like other multiples of 12, this is a common number of dimensions in

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

the financial domain, because a year has 12 months, and the same phenomenon will occur if the generator is used for pricing other derivative securities.

We have also presented a method for avoiding wrong convergences. Combined with the antithetic variable technique, ran1() can give the correct value with a good convergence speed.

We have observed that statistical tests used to measure the randomness of pseudo-random numbers have difficulty in detecting the anomaly of ran1(). The effect of another anomaly found in the experiment is not known.

References

[1] M. Broadie, and P. Glasserman: Pricing American-style securities using simulation. Working paper, Columbia University, 1995.

[2] A. M. Ferrenberg, D. P. Landau and Y. J. Wong: Monte Carlo simulations: hidden errors from "good" random number generators. *Physical Review Letters,* **69** (23) (1992) 3382-3384.

[3] J. C. Hull: *Options, Futures, and Other Derivative Securities, Second Edition* (Prentice Hall International, 1993).

[4] D. E. Knuth: *The Art of Computer Programming, Second Edition, Vol. 2, Seminumerical Algorithms* (Addison-Wesley, 1981).

[5] W. J. Morokoff and R. E. Caflisch: Quasi-random sequences and their discrepancies. *Journal on Scientific Computing,* **15** (6) (1994) 1251-1280.

[6] R. Motwani and P. Raghavan: *Randomized Algorithms* (Cambridge University Press, 1995).

[7] H. Niederreiter: *Random Number Generation and Quasi-Monte Carlo Methods,* CBMS-NSF Regional Conference Series in Applied Math., No. 63, (SIAM, 1992).

[8] S. H. Paskov: Computing high dimensional integrals with applications to finance. Technical Report CUCS-023-94, Columbia University, 1994.

[9] W. H. Press, S. Teukolsky, W. Vetterling and B. Flannery: *Numerical Recipes in C, First Edition* (Cambridge University Press, 1988).

[10] I. M. Sobol: The distribution of points in a cube and the approximate evaluation of integrals. *USSR Computational Mathematics and Mathematical Physics,* **7** (1967) 86-112.

[11] A. Tajima, S. Ninomiya and S. Tezuka: On the anomaly of ran1() in Monte Carlo pricing of financial derivatives. *1996 Winter Simulation Conference Proceedings,* (1996) 360-366.

[12] S. Tezuka: *Uniform Random Numbers: Theory And Practice* (Kluwer Academic Publishers, 1995).

[13] J. A. Tilley: Valuing American options in a path simulation model. *Transactions of the Society of Actuaries,* **45** (1993) 499-519.

[14] B. Moro: The full monte. *RISK,* **8** (2) (1995) 57-58.

Akira TAJIMA: `tajima@trl.ibm.co.jp`
Syoiti NINOMIYA: `ninomiya@trl.ibm.co.jp`
Shu TEZUKA: `tezuka@trl.ibm.co.jp`
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa 242-8502, Japan

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.