Asian Journal of Computer Science and Technology
ISSN: 2249-0701 Vol.8 No.2, 2019, pp. 1-5

Evaluation of Random Number Generator Functions Using Statistical Analysis

Rajashree Chaurasia
Department of Computer Engineering, Guru Nanak Dev Institute of Technology
Directorate of Training & Technical Education, Government of NCT of Delhi, India
E-Mail: rajashree.chaurasia@gmail.com

Abstract **- Most programming languages have in-built functions for the sole purpose of generating pseudo-random numbers. This manuscript is aimed at analyzing the appropriateness of some of these in-built functions for some basic goodness-of-fit statistical tests for random number generators. The document is divided into four sections. The first section gives a broad introduction about randomness and the methods of generation of pseudo-random numbers. Section two discusses the statistical tests that were employed for testing the built-in library functions for random number generation. This section is followed by an analysis of the data collected for the various statistics in the third section, and lastly, the fourth section presents the results of the data analysis.**
Keywords: **Random Numbers, Statistical Analysis, Random Number Generation, Programming Language, Built-in Functions, C/C++, C#, Java**

I. INTRODUCTION

Random numbers have uses in any area where there arises a need to make an un-biased decision or choice. For instance, in ancient times in some countries, leaders and officials were not elected by ballot vote; rather, they were chosen using random procedures like drawing out names from a lot. The most noticeable example of randomness has been the process of genetic mutation and evolution where random sequences of DNA or RNA nucleotide bases are altered depending on some external conditions. Today, random numbers are used in a large number of fields [1]like simulation, mathematics and statistics, decision making, biological sciences, testing, network security and cryptography, communication systems, gaming and gambling, etc., etc.

Random numbers are generally classified as either true-random or pseudo-random. True random numbers are generated using physical or mechanical means; thus, these numbers are produced such that they are completely independent of other generated random numbers. This is why they are known as truly random. Wherever high quality randomness is desired, we use physical or mechanical generators. For other purposes, we use computational methods to generate random numbers and these are pseudo-random in nature. This means that the next random number in the sequence is partially dependent on the previously generated random number.

The broad categories of methods [1] [3] for pseudo-random number generation are

1. Linear Congruential method
2. Quadratic Congruential method
3. Inversive Congruential method
4. Multiplicative Congruential method

By far, the most popular random number generators in use today are special cases of the linear congruential scheme, introduced by D. H. Lehmer in 1949 [4].

A. Linear Congruential Method

Four magic numbers [1] are selected such that

$X_{n+1} = (aX_n + c) \bmod m,\ n \geq 0$

where m is the modulus,
a is the multiplier,
c is the increment,
X_{n+1} is the next random number and
X_n is the previous random number

The limitation of this scheme is that the sequence of numbers repeats itself as the maximum period can only be of size m. Therefore, we need to carefully choose the values of the modulus, the multiplier and the seed or the starting number in the sequence (generally denoted by X_0).

B. Quadratic Congruential Method

The generalized equation in this scheme is

$X_{n+1} = (dX_n^2 + aX_n + c) \bmod m$

where m is the modulus,
a,d are multipliers,
c is the increment,
X_{n+1} is the next random number and
X_n is the previous random number

The limitation of overflow which was present in the linear congruential method is removed in the quadratic scheme.

C. Inversive Congruential Method

The generalized equation under this method is

$X_{n+1} = (aX_n^{-1} + c) \bmod p$

where p is the modulus (p is also prime),
a is the multiplier,
c is the increment,
X_{n+1} is the next random number and
X_n is the previous random number

This method was suggested by Eichenauer and Lehn.

D. Multiplicative Congruential Method

George Marsaglia [5] suggested that the linear congruential method can be converted into a multiplicative form. His method defines the generalized equation as under:

$X_n = (X_{n-24} * X_{n-55}) \bmod m, n > 55$

where m is the modulus (m is also a multiple of 4),
X_n is the next random number and
X_{n-24} is the 24^{th} previous random number
X_{n-55} is the 55^{th} previous random number

II. GOODNESS OF FIT TESTS

There are many desirable characteristics of a good random number generator algorithm. For instance, the basic criterion for a random sequence generator algorithm is a long period. But this criterion is not good enough to judge the randomness of the generator algorithm. Other criteria like absence of a discernible pattern, the range of values that is expected to be observed, the nature of the distribution functions for the random sequence, etc. are also important. The theory of statistics provides us with some quantitative measures for randomness. There are an infinite number of such tests available but this manuscript discusses only a few of them that are the more standard or what we call as basic and trusted statistical tests.The author has performed extensive analysis of various random number generator functions of programming languages for the following statistical tests.

A. Chi-Square Test

The chi-square test (χ^2 test) [1] is perhaps the best known of all statistical tests, and it is a basic method that is used in connection with many other tests. It is based on a comparison between the empirical distribution function and the theoretically expected distribution.

We have k categories in this scheme and each observation falls under any one of these categories. For $O_i$the number of observed occurrences in a particular category, let E_i denote the number of expected occurrences, we calculate the chi-square statistic as under:

$$V = \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}$$

We count the number of observations falling into each of k categories and compute the quantity V. Then V is compared with the numbers in the standard chi-square statistic table can be found at [1] [2] for different degrees of freedom (which is one less than the number of categories, k). If V is less than the 1% entry or greater than the 99% entry, we reject the numbers as not sufficiently random. If V lies between the 1% and 5% entries or between the 95% and 99% entries, the numbers are suspect; if V lies between the 5% and 10% entries, or the 90% and 95% entries, the numbers might be almost suspect. The chi-square test is frequently done at least thrice on different sets of data, and if at least two of the three results are suspect the numbers are regarded as not sufficiently random [1].

B. Frequency or Equidistribution Test

Frequency test [3] checks if the generated numbers are equally distributed and this test is used in collaboration with the chi-square test. Here, we count the number of occurrences of each number generated in the specific range and compute the chi-square statistic by taking the degrees of freedom as one less than the range and E_i as inverse of the maximum range.

C. Poker Test

The poker test [1] [7] takes random numbers in groups of k (successive) figures from the generated sequence and categorizes them as r different values. A chi-square test follows where the probability is calculated using the formula

$$p_r = \frac{d\,(d-1)\ldots(d-r+1)}{d^k} \left\{ {k \atop r} \right\}$$

where $\left\{ {k \atop r} \right\}$ is Sterling's number of the second kind

d is the maximum value that the random number can take.

D. Kolmogorov-Smirnov Test

The chi-square test is applicable in cases where the randomly generated numbers fall into a discrete number of categories. But there in a situation where the categories that the numbers fall into are continuous or infinite in nature, the chi-square test fails to be appropriate. This is especially the case with floating point or real numbers, i.e. numbers that range between 0 and 1. For such circumstances, we employ the Kolmogorov-Smirnov or KS test [1]. The random number generators distribution function $F_n(x)$ is compared to the expected distribution function F(x) and two statistics are computed:

$$K_n^+ = \sqrt{n} \max_{-\infty<x<+\infty} (F_n(x)-F(x))$$

$$K_n^- = \sqrt{n} \max_{-\infty<x<+\infty} (F(x)-F_n(x))$$

K_n^+ measures the greatest amount of deviation when F_n is greater than F, and K_n^-measures the maximum deviation

when $F_n$is less than F. Like the chi-square table, we can look-up the values for the above two statistics in the KS table (found at [1]).

III. DATA ANALYSIS

All implementation for the goodness-of-fit statistics has been done in C/C++ programming language. Each of the built-in library functions (rand and random in Turbo C++ 3.0 and GCC compiler, nextInt and nextFloat in Java, Next and Next Double for Microsoft's Visual C++ and Visual C# 2005) have been tested for each of the four standard goodness-of-fit statistical tests viz. Chi-square, Frequency, Poker and KS test.

In all the tabular data that follow, numbers in red colour correspond to 'rejected/failed result'; numbers in orange colour signify 'suspected result' and numbers in green colour represent 'satisfactory result'.

A. Chi-square Test on Various Compilers

For Borland's Turbo C++ compiler version 3.0, both the functions: rand and random were tested by the author for the chi-square statistic. The function rand was paired with srand and random with randomize, where srand and randomize are both random seed generators for the random generator functions - rand and random.

Two sets of data were taken for random: one for 1000 random integers between 0 and 1000 (including 0 and excluding1000), and the other for 1,000,000 random integers between the same range. For rand, the sets were 1 million random integer numbers in the range [0, 1000) and 1 million random integer numbers in the range [0, 10000). For Java, VC++, VC# and GCC compilers, the same nature of data-sets was used: 1 million random integer numbers in the range [0, 1000) and 1 million random integer numbers in the range [0, 10000).

TABLE I DATA FOR CHI-SQUARE STATISTIC

Compiler	Range, Numbers	Set 1	Set 2	Set 3
Turbo C++ (random)	1000,1000	15.940001	13.380000	3.120000
	1000,1000000	13.962389	11.651489	20.523750
Turbo C++ (rand)	1000,1000000	181.384811	142.377869	143.787628
	10000,1000000	16761.384766	16657.951172	16628.130859
Java	1000,1000000	13.428130	6.664810	9.033429
	10000,1000000	6.194030	7.504190	10.712070
VC++	1000,1000000	4.052150	4.532610	4.968510
	10000,1000000	6.710070	9.88089	6.563169
VC#	1000,1000000	5.684190	6.343290	8.370410
	10000,1000000	5.953390	5.667530	3.678570
GCC	1000,1000000	14.226200	11.601140	7.477960
	10000,1000000	5.780420	8.463280	6.683540

The result of the Chi-square statistic analysis for the function random in Turbo C++ 3.0 is mostly 'suspect' or 'reject'. This shows that random did not pass the χ^2 goodness-of-fit test.

Similarly, tests were conducted for the rand function and it was found that rand failed terribly in this case. All the probabilities of the chi-square statistic turn out to be in the less than 0.01% range which is extremely rare and thus, not at all a characteristic of random behavior.

This is so because rand has a period of only 232 which is not large enough for most practical applications of random numbers. In this context, random seems to be a slightly better choice. All the other compilers generated satisfactory results and passed the chi-square test. The functions available for random number generation in these languages global randomness over the rand and random functions are clearly superior generators and surpass the property of available in standard C libraries.

B. Frequency Test on Various Compilers

For Borland's Turbo C++ compiler version 3.0, both the functions: rand and random were tested by the author for the chi-square equidistribution statistic. The function rand was paired with srand and random with randomize, where srand and randomize are both random seed generators for the random generator functions - rand and random. Two sets of data were taken: 1 million random integer numbers in the range [0, 1000) and 1 million random integer numbers in the range [0, 10000).

Degrees of freedom are taken to be 999 for numbers in the range 0 to 999 and 9999 for numbers in the range 0 to 9999. The chi-square values for such high degrees of freedom can be found at [6].

For Java, VC++, VC# and GCC compilers, similar nature of data-sets were used: 1 million random integer numbers in the range [0, 1000) and 1 million random integer numbers in the range [0, 10000).

All the above compilers generated satisfactory results and passed the frequency test. The functions available for random number generation in these languages are clearly superior generators and surpass the property of local randomness over the rand and random functions available in standard C libraries.

TABLE II DATA FOR FREQUENCY STATISTIC

Compiler	Range, Numbers	Set 1	Set 2	Set 3
Turbo C++ (random)	1000,1000	1102.924805	1145.662354	1047.042358
	1000,1000000	28925.566406	28718.972656	28191.701172
Turbo C++ (rand)	1000,1000000	1171.201050	1214.511475	1170.268921
	10000,1000000	29055.613281	28875.595703	28508.636719
Java	1000,1000000	930.928772	987.450378	948.042542
	10000,1000000	9865.218750	9877.512695	9843.210938
VC++	1000,1000000	968.678101	991.907593	981.381165
	10000,1000000	9968.183594	10039.478516	9799.747070
VC#	1000,1000000	968.224304	949.746521	965.967346
	10000,1000000	10089.733398	9900.225586	9849.294922
GCC	1000,1000000	949.483398	1006.139282	1018.885254
	10000,1000000	9921.503906	9961.891602	9912.098633

C. Poker Test on Various Compilers

The poker test is used in combination with the chi-square test. For the range of integer random numbers generated, the observed occurrence of each hand is compared with its expected frequency. The poker test is the worst and least comprehensive of all tests. To create a simpler version of this test, the author counted the number of distinct values, which is called r, in the set of five categories as below:
5 values = all different;
4 values = one pair;
3 values = two pairs, or three of a kind;
2 values = full house, or four of a kind;
1 value = five of a kind.

Thus, r ranges from 1 to 5. The range of random numbers generated is also from 1 to 5 for all compilers. Four sets of data for 100, 1000, 10000 and 100000 random numbers have been tested by the author for each compiler.

TABLE III DATA FOR POKER STATISTIC

Compiler	N=100	N=1000	N=10000	N=100000
Turbo C++ (random)	9.817709	0.442708	5.914322	0.558047
Turbo C++ (rand)	8.020834	13.567709	5.913803	6.764818
Java	2.9427083	7.0677090	5.1885423	5.1014063
VC++	1.640625	3.450520	0.906510	2.962344
VC#	6.82291667	1.56510446	4.8776053	5.18893229
GCC	7.109375	4.38541564	5.1106773	3.17302083

As is apparent from the data above, both the functions random and rand fail for the poker test as well. Here we have only 4 degrees of freedom as there are only 5 categories for partitioning the data sets. All the other compilers generated satisfactory results and passed the poker test. The functions available for random number property of associative correlation over the rand and random functions available in standard C libraries.

D. KS Test on Various Compilers

This test is performed on uniform random numbers in the range (0, 1). Three sets of data with 10 random numbers (floating point) between 0 and 1 were tested for in each compiler.

As is apparent from the data above, both the functions random and rand (in Turbo C++ 3.0 compiler) fail for the KS test as well. All the other compilers generated satisfactory results and passed the KS test.

The functions available for random number generation in these languages are clearly superior generators and surpass the property of both global and local randomness over the rand and random functions available in standard C libraries.

TABLE IV DATA FOR KS STATISTIC

Compiler	KS	Set 1	Set 2	Set 3
Turbo C++ (random)	$K10^+$	0.92295500	1.403997524	1.522886513
	$K10^-$	1.23918277	1.087769756	1.206658747
Turbo C++ (rand)	$K10^+$	1.70605828	1.032202213	0.843528079
	$K10^-$	1.38983051	1.348429979	1.159755845
Java	$K10^+$	0.70666596	0.418513407	0.778027980
	$K10^-$	0.39043820	0.458627342	0.643853171
VC++	$K10^+$	0.77027315	0.388148650	0.897749607
	$K10^-$	0.45404538	0.604176229	0.581521841
VC#	$K10^+$	0.73304877	0.657696044	0.530550232
	$K10^-$	0.65820514	0.565283226	0.560006243
GCC	$K10^+$	0.78439032	0.450447479	0.699597011
	$K10^-$	0.71067763	0.752353289	0.719702772

IV. CONCLUSION

Extensive research was conducted by the author to put to test some of the built-in functions available for generating random numbers in some commonly used programming languages viz. Java, C++, C#, C and their compilers like the Borland compiler version 3.0 for C/C++, the GNU GCC compiler for C, the Visual C++ 2005 compiler for C++, the Visual C# 2005 compiler for C# and the Javac compiler version 1.6.0 Update 31 for Java. These built-in methods were tested by four standard goodness-of-fit statistical tests for random behaviour (viz. Chi-Square test, Equidistribution test, Partition or Poker test and the Kolmogorov-Smirnov test) using datasets of 1 million integer random numbers between specific ranges and 10 floating point random digits with high precision, between the range 0 and 1. The results of the data collected during this research clearly shows that the functions rand and random in Turbo C++ 3.0 library failed all the four goodness-of-fit tests. All other compilers generated satisfactory results and passed each test. This implies that due to the very short period length of the functions, the 1 million random numbers generated using the same did not exhibit agreeable random behaviour. Thus, it can be said that these generator functions have a poor performance when compared to the library functions of the other compilers namely, Java, VC++, VC# and GCC. The frequency test, being the most comprehensive of all tests gave striking results against Turbo C++ functions; they failed badly in the equidistribution test. This is because the frequency test compares the occurrence of each possible random number in the specified range with what its expected frequency should be and then applies the chi-square statistic. On the contrary, the poker test was the worst among the group of test employed. This is because when the range of random numbers generated increases, the probabilities of the categories of all fives, four-of-a-kind, three-of-a-kind and two-pair also decrease proportionately. Thus, the only category for which the occurrence can be counted is 'bust' or all-different. Using this category alone does not give a proper estimate on the goodness of the randomness property. This makes the poker test applicable for very small ranges only. For this purpose, the poker test was carried out for random numbers in the closed range of [1, 5].

REFERENCES

[1] E. Knuth, "The Art of Computer Programming", *Semi-numerical Algorithms*, 3rd ed., Boston, MA, USA: Addison Wesley Longman Inc., Vol. 2, 1997.

[2] M. Abramowitz and I. A. Stegun (Eds.), *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, Washington, D.C., USA: Government Printing Office, Table 26.8, 1964.

[3] N. Deo, *System Simulation with Digital Computer*, NJ, USA: Prentice Hall Inc., 2006.

[4] D. H. Lehmer, "Mathematical methods in large-scale computing units", In *Proc. 2nd Symposium on Large-Scale Digital Calculating Machinery*, Cambridge, MA: Harvard University Press, pp. 141-146, 1951.

[5] G. Marsaglia, "A Current View of Random Number Generators", Keynote Address, *In Proc. of the Computer Science and Statistics*: 16th Symposium on the Interface, Atlanta, 1954.

[6] Fourmilab Switzerland Chi-square calculator. [Online]. Available: http://www.fourmilab.ch/rpkp/experiments/analysis/chiCalc.html, 2012.

[7] M. Rütti, "A Random Number Generator Test Suite for the C++ Standard", *Diploma Thesis*, Institute for Theoretical Physics, ETH Zürich, March 2004.