A Better Random Number Generator for Apple's Floating Point BASIC

Cem Kaner & John R. Vokey

Published in MICRO, June 1984, p. 26-35.

Random number generators (RNGs) are functions which produce pseudo-random numbers. Usually, the numbers produced are fractions between 0 and 1. Ideally, every fraction a computer language can represent will be as likely to be generated as every other by that language's RNG, and the order of numbers will be unpredictable to the user. Slightly more formally, the RNG should produce sequences of numbers which, so far as standard statistical tests can tell, behave in the same way as "truly random" number sequences would behave. RNGs are used to simulate imperfectly predictable real life events (computer games use them, for example), to provide test data for input to complex computer programs (often exposing bugs that a series of test inputs missed), to evaluate mathematical functions or solve equations for which no theoretical solution exists, or similarly, to provide estimates of a solution against which a new theoretical solution can be checked. Randomization, or randomly rearranging the order of things, is the basis of a wide class of increasingly popular, very sensitive statistical tests. Randomization of the order of events in experiments is a necessary part of the design of every experiment that we have run. The better the generator, the better the simulation, solution, test, or experimental control.

Most implementations of high level computer languages provide an RNG in their function library. The RND function of Applesoft, Apple's floating point BASIC, is typical of those we've seen on small systems. The language's reference manual describes RND as a source of random numbers. The reference manual provides no evidence whatever that this claim should be believed, nor any warning that it should be taken with a grain of salt. The function, when subjected to standard statistical tests, fails them miserably. We are not singling Apple out for criticism. Manuals which admit to low-grade RNGs are nearly as rare as language implementations which provide an RNG worthy of the name. In this article, we present an assembly language program, interfaced with Applesoft as the USR function, which provides three independently addressable RNGs. We also present test data which suggest that the RNGs will be suitable for most applications.

The RNG Algorithm

There are many ways to produce pseudo-random numbers, a few of which work reasonably well. Donald Knuth's excellent 178-page chapter on RNGs describes quite of few of all varieties. We used what is called a *linear congruential generator*. If R_n was the last number produced, and **a, c,** and **m** are constants, then

$$R_{n+1} = (a\ R_n + c) \text{ modulo } m.$$

To obtain a number modulo **m,** divide it by **m**, but keep the reminder rather than the quotient. For example, 13 mod 10 = 23 mod 10 = 972863 mod 10 = 3. The generator does not produce "truly random" numbers (no software RNG can) because it is possible, given knowledge of **a, c,** and **m** to predict the next number from the last. However, if a and c are properly chosen, and m is reasonably large, a person who did not know the formula, or even one who knew it but didn't have a high precision calculator handy, would be very hard-pressed to predict the next number.

Selection of the RNG's Parameters

It is easy to find values of a and c which will ensure that the RNG will produce every number between 0 and $m-1$ before the sequence starts to repeat itself. Eventually, no matter what a, c, and m are, the series must repeat. How long it goes without repetition is called the *period*, and so, for proper a and c, the period is m. Note that at this point, our RNG is producing integers. To get fractions, just divide these integers by m. Now, suppose your computer reserved 32 bits for the digits of any number, as the Apple does. Then, by taking $m=2^{32}$, you would obtain every bit pattern that could be stored for a number. Unfortunately $m=2^{32}$ will not yield every fraction that the Apple can store, because the Apple uses an extra byte per number to hold an exponent. This allows the representation of thousands of different, very small numbers, say near 10^{-38}, whereas working with fractions of the form $R_n / 2^{32}$, we can only produce one number in this region, namely zero. Tiny fractions in floating point languages are under-represented by congruential generators: many fractions the language can work with cannot be generated. We can alleviate the problem somewhat, and increase the period, by increasing m to 2^{40}. Not every possible fraction will be generated—R_n would have to be 17 bytes long for that and the RNG would be very slow—but R_n will take on 1,099,511,627,776 different values, which should be enough for most purposes.

Having settled on $m=2^{40}$, we need to choose a and c. Choosing values which ensure a period of length m is only part of the story. These are easy to find. For example, $R_{n+1} = R_n+1$ will generate every possible value, but this is hardly random. The apparent randomness of the sequence is a function of the ordering of the numbers, and this is where most RNGs, including all linear congruential generators, have at least some problems. We can think of the ordering problem by thinking of sequences of the form R_n, R_{n+1}, R_{n+2}, … R_{n+k}. Consider pairs first. There are m^2 possible pairs of fractions between 0 and $m-1/m$, but a generator of period m can only yield $m-1$ different pairs (R_n, R_{n+1}). *Which* $m-1$ pairs is the critical question. In the case of $R_{n+1} = R_n+1$, a graph of R_n along one axis and R_{n+1} along the other would show a single straight line. A truly random sample of $m-1$ cases from the possible m^2 would have points scattered all over the graph. *All linear congruential generators will produce graphs which show patterning, and that patterning will always be a set of parallel lines. The trick is to find a generator which produces as many of these lines as possible, which as few points on each line as possible.* The result will be a more even coverage of the m^2 possible pairs. In three dimensions, a graph of points (R_n, R_{n+1}, R_{n+2}) will always show parallel planes. In this case, only $m-2$ possible m^3 possible triplets are produced, so coverage of the space is more sparse than for two dimensions, and the problem of patterning is potentially more serious. In higher dimensions (longer sequences) we get parallel hyperplanes.

Our goal, then, is to find values of a and c which produce as many lines, planes, and hyperplanes as possible, and which space them out as evenly as possible. This can be translated into a goal of minimizing the maximum distance between any two lines (planes, etc.). Let $1/v_2$ be the maximum distance between any adjacent pair of lines in a graph of R_n against R_{n+1}, let $1/v_3$ be the largest distance between pairs of parallel planes in the graph of R_n by $R_{n+1}$by R_{n+2}, and so on. Our goal is to maximize v_2, v_3, v_4, v_5, and v_6. We stop at 6 because if these values are good, higher dimensional sequential interactions are probably unimportant. According to Knuth, we would be pretty safe stopping at 4.

The v_i's, for linear congruential generators, can be determined using a method first proposed by Coveyou and MacPherson in 1965, which is based on the finite Fourier transform. The mathematics underlying this test, the *Spectral Test* of an RNG, are beyond the scope of this article, but they are well described by Knuth. To compute the v_i's, we used Knuth's Algorithm S, which requires high precision *integer* arithmetic, which Apple's PASCAL is well suited to perform. This algorithm, taking only a and m for input (c is irrelevant), quickly determines the values of the v_i's for the output of the generator across its entire period. To choose the multipliers for the three generators, we computed v_i's for about 30,000 values of a until three suitable values were found. For archival purposes, we list the values of the v_i's in Table 1, which also gives the parameters a and c for each generator, and some other information discussed below. These values compare favorably with the values of standard congruential generators on large (e.g., IBM 370) computers (see Knuth, pp. 102-104). The more interesting values in the table are the values of u_2, u_3, u_4, u_5, and u_6. These essentially normalize the v_i's relative to the best values obtainable for a given m. Knuth defines a value of v_i greater than 0.10 as a "pass" of the Spectral Test, and notes that every generator known to be bad fails this test. He defines a "pass with flying colors" as $v_i > 1.0$. Our smallest value being 2.37, the three generators are acceptable by this criterion.

Generator	**X**	**Y**	**Z**
Multiplier	27182819621	8413453205	31415938565
Additive Constant	3	99991	24607
V_2	982974962600	1112748837514	908473954394
V_3	72937326	103184754	79566866
V_4	1023550	805970	1036504
V_5	58786	60670	59710
V_6	9916	8142	11636
V_2	2.81	3.18	2.60
V_3	2.37	3.99	2.70
V_4	4.70	2.91	4.82
V_5	4.01	4.34	4.17
V_6	4.58	2.54	7.40

Readers familiar with statistical techniques may wonder why we didn't use a more traditional test of sequential clustering, the *Serial Test*. Since we will report Serial Test results later for a subsequence of the period (the first 850,000 numbers from each generator), we should describe the test. Suppose you split the range from 0 to 1 into 10 equal subranges, 0 to 0.10, 0.10 to 0.20, 0.20 to 030 and so on. If you generate a sample of 10,000 numbers, you can count how many are in each subrange. A random source would produce about 1000 for each, and this can be compared to the number that the RNG produces. Similarly, you can consider pairs (R_n, R_{n+1}). On average, 100 pairs will have both R_n and R_{n+1} less than 0.10, another 100 will have $R_n<0.10$ and $0.10<R_{n+1}<0.20$ and so on. There are 1000 possible types of triples, (R_n, R_{n+1}, R_{n+2}) and we'd expect 10 of each in this example. The traditional test used to examine the difference between the number of each number, pair, triple, etc. that the RNG produces and the number of each expected is called the Serial Test, developed by I.J. Good in 1953. Conceivably, such a test could be run on the entire output of the RNG (all trillion-plus numbers). For such a large sample, this test is known to be extremely sensitive to deviations from randomness. A very important result on the Spectral Test was obtained by Neiderreiter (see Knuth): Any RNG that passes a full period Spectral Test will also pass a full period Serial Test. Thus, by using the Spectral Test to determine the three values of **a**, we ensure that the RNGs pass both the full period Spectral Test and the full period Serial test. For parameter selection, then, all that is left is to choose the values of **c**.

While the additive constant is of no importance for the Spectral Test, it does influence the value of another traditional test of ordering, the *Serial Correlation Test.* You can think of the serial correlation, lag **k**, as measuring the degree to which the relationship between $\mathbf{R_{n+k}}$ and $\mathbf{R_n}$ can be described as linear. A value of zero indicates no linear relationship between the random number produced now by the RNG and the value that it will produce **k** calls from now. A value +/- 1 indicates a perfect linear relationship and an atrocious RNG. Knuth's Theorem K gives a method to establish upper and lower bounds on the correlation across the entire period. We applied it to test several additive constants, for each of the RNGs, for serial correlations lag 1 through lag 20. There are thus 60 correlations, 20 for each generator. For the values of **c** chosen, the largest correlation lies somewhere between –0.00000001135 and 0.00000000569. The second worst case lies between –0.00000000038 and 0.00000000072. These seem close enough to zero for most applications.

In summary, the modulus value of 2^{40} resulted from a compromise between considerations of speed and space on the one hand, and of period length and tiny value representation on the other. The critical full period tests from here were tests of sequential relationship. Equal frequency is, of course, a major criterion of randomness, but this entered into our parameter selection only insofar as values of **a** and **c** which would not guarantee equal frequently were rejected automatically. The parameter selection was determined, for each generator, by performance on the major full period tests of sequential relationship.

Empirical Tests of the RNGs

Full period tests only tell us about the performance of the RNG across the entire period. They do not guarantee that subsequences will be good. It could be that a strong trend in, say, the first 100,000 values will be counterbalanced by a reverse trend in the next 100,000 and soon. Since no application we know of would use the full trillion number period, the only way to be confident about quality for actual use is to examine the RNG's subsequence behavior.

To do this, we ran a number of standard statistical tests on the output of each generator, examining the output in batches (i.e. subsequences) of 1000 to 10,000. For each test, sampling started at the (same) starting values of the generators. Many users will only need these first few hundred thousand numbers, so these should be the ones most carefully examined.

1) Serial Tests

We defined these tests in the discussion of the Spectral Test, above. Samples of 10,000 numbers were tested for simple frequency (number of $\mathbf{R_n}$'s < 0.10, between 0.10 and 0.20, and etc.), and for clustering of pairs and triples. Eighty-five batches were examined and the 85 results, for each test and each generator, were compared to the distribution of results we would expect from a random source, using the *Kolmogorov-Smirnov Test.* All three generators passed the simple frequency ($p > 0.10$) and triples ($p > 0.20$) tests. The generators listed as X and Y in Table 1 passed the doublets test, but Z did not. The problem with Z, which is seen again below, is that it does too well on these tests. If you test a truly random source many times, it will sometimes fail a test of randomness and sometimes only marginally pass it. Z's performance was sometimes this poor, but not often enough to mimic a random source ($0.05 > p > 0.02$). Since nothing can be "more random" than a "truly random" source, this must be a flaw in Z. It should be realized, though, that these tests are *very* sensitive to minor deviations from random source behavior when such huge (850,000) sample sizes are involved. Z does not perform ideally, but this performance is a far cry from bad.

We attempted to test the Applesoft RND in the same way for comparison. On the first batch of 10,000, RND failed only the triplets test. Unfortunately, between the 10,000th and the 20,000th number generated, RND fell into an endless loop, repeating itself every 202 numbers. Tests beyond this point, then, were meaningless, though perhaps no more so than the application of the name RND to a generator of period 202.

2) Frequency Tests

The simple frequency test in the Serial Test is a standard test on its own, called the *Chi-Square Test.* Equal frequency is of major concern for sub-sequence testing. The fact that all possible numbers will eventually appear is no guarantee that they will come in a reasonable order. It all too often turns out that an RNG yields far too few, then far too many small (large, whatever) numbers. Another frequency test requires no grouping of the numbers and is more sensitive to departures from randomness than the Chi-Square Test. This test, the *Kolmogorov-Smirnov Test* (KS test), compares the proportion of numbers generated which lie below any given (every) number between 0 and 1 with the proportion we'd expect from a random source. A hundred such tests, for each generator, were run on batches of 1,000 numbers, and the KS test was then used to compare the distribution of KS values from these 100 tests per generator with the distribution a truly random source would give. X and Y passed ($p > 0.20$). Z failed, even though it had already passed the Chi-Square Test. The problem with Z, as before, is that its test performance is too good, too often ($0.05 > p > 0.02$). This is a most unusual problem for a RNG and, given the sensitivity of the test, it is not a large one, but it should be noted.

3) Runs Tests

A run up is a succession of increasing numbers (e.g., 0.1, 0.2, 0.3) which ends when the next number generated is lower than the last. A run down is similarly defined. The number of increasing or decreasing numbers produced by the RNG is the length of the run. Tests of how many runs, and how many runs of each length, are further tests of sequential trend in the RNG. Both of these tests were run, for each generator, on a sample of 50,000 numbers. All generators passed them handily.

In sum, X and Y passed all tests, theoretical (full period) and empirical (sub-sequence). Z's sub-sequence performance is less good, and we do not recommend its use as a stand-alone RNG. X and Y appear sufficiently random for most needs. Note that they are not perfect. We still have the parallel lines and planes problem, common to all linear congruential (and many other) generators, even though we have minimized it. For very precise simulations, this is not good enough. However, if more than one RNG is available (which is why we provide three), we can do better than this.

Combination of RNGs

The graph of the last number generated, $\mathbf{R_n}$, against the number generated this time, $\mathbf{R_{n+1}}$, shows a family of parallel lines when all pairs ($\mathbf{R_n}$, $\mathbf{R_{n+1}}$) are plotted. This is the parallel lines problem. If our goal is to break down this linear structure, as we must do to mimic the random structure produced by a truly random source, why not just randomly rearrange the order of the numbers generated by the RNG as it generates them? This is George Marsaglia's insight, and in practice it works out very well. Here's an example of the procedure for doing this. Generate 100 values from X and store them in a matrix, say XRAN[I]. Now sample a value from generator Y, and use this value to determine which is the next value you'll take from XRAN[I], i.e. RANDOM=XRAN[Y*100]. Replace the sampled value of XRAN with a new one from generator X and you're done. A sequence of numbers, RANDOM[I], produced in this way will still have the same good subsequence frequency properties as X does, but the last remnants of sequential patterning from X typically disappear. Knuth gives examples of quite poor generators which perform very well when combined in this manner. Any combination of X, Y, and Z should be quite good, though the cautious user might restrict Z to the role of the selection generator (as Y was above) due to its too equal subsequence frequencies.

A second approach is to sample from X, then Y, then Z in turn. This triples the period and can destroy the lower-dimensional patterns (in our case, the parallel lines problem), but it will not do so for all generators combined in this way. In fact, Lewis' multi-RNG Theorem (pp. 18-19) states that if any multiplier in a bank of equal period RNGs is near the square root of **m**, the problem will return with a vengeance. We restricted selection of multipliers for X, Y and Z to values far from the square root of **m** (1,048,576) in order to allow this form of generator combination. For these generators, according to Lewis, this technique should be very effective.

The last approach we'll mention is to use one generator (e.g. Z) to decide which of the other two will be sampled from this time. This only doubles the period of the resulting generator (if you need 3 trillion numbers, use a different RNG), but it does randomize the order of sampling, which is not done above.

It seems probably important for each solution above that the different generators' outputs be unrelated. Otherwise, replacing a value of X with one from Y, for example, might make little difference. Our final test of the generators involved computing the correlation (measure of linear relationship) between X and Y, X and Z, and Y and Z. A hundred correlations were taken, on samples of 1,000 numbers per generator. All were reasonably low. The averages were 0.0003 for X and Y, 0.0038 for X and Z, and –0.0017 for Y and Z, which should be low enough to all combinations.

Using the RNG

The rest of the paper describes an assembly language implementation of this algorithm. Almost no one uses 6502 Assembler any more, or needs to know the details of the hooks in Applesoft for extending the language, so I've deleted that material. If you need it, write me for a photocopy. Kaner@kaner.com

References

Edgington, E.S., *Randomization Tests*, Marcel Dekker, 1980.

Good, I.J., "The serial test for sampling numbers and other tests for randomness," *Proceedings of the Cambridge Philosophical Society*, 1953, *49*, 276-284.

Kendall, M.G. & Stuart, A., *The Advanced Theory of Statistics, Volume 2: Inference and Relationship* (third edition, 1973), *Volume 3: Design and Analysis, and Time Series* (third edition, 1975). Hafner Press.

Knuth, D.E., *The Art of Computer Programming, Volume 2: Seminumerical Algorithms,* Addison-Wesley, 1981. (Note that the version of Algorithm S that we used is the version found in the first edition of this book, 1969).

Lewis, T.G., *Distribution Sampling for Computer Simulation.* Lexington Books, 1975.

Marsaglia, G., “Random number generation” in A. Ralston, *Encyclopedia of Computer Science*, Van Nostrand Reinhold, 1976, 1192-1197.

For an introduction to using random numbers on home computers, see Albrech and Firedrake’s book, *The Mysterious and Unpredictable RND.*