Standard Random Number Generation for MBASIC

R. C. Tausworthe
DSN Data Systems Section

This article presents and analyzes a machine-independent algorithm for generating pseudorandom numbers suitable for the standard MBASIC system. The algorithm used is the "polynomial congruential" or "linear recurrence modulo 2" method devised by the author in 1965. Numbers, formed as nonoverlapping, adjacent 28-bit words taken from the bit stream produced by the formula $a_{m+532} = a_{m+37} + a_m$ (modulo 2), will not repeat within the projected age of the solar system, will show no ensemble correlation, will exhibit uniform distribution of adjacent numbers up to 19 dimensions, and will not deviate from random runs-up and runs-down behavior.

I. Introduction

The first MBASIC random number generator (Ref. 1), implemented on the Univac 1108, used a linear congruential method, $x_{n+1} = 5^{15}x_n$ (mod 2^{35}), followed by normalization to the range (0,1). This generator was used probably because it was already available in the U1108 statistics package. Empirical tests by users, however, later proved that the generator possesses very nonrandom correlation properties indeed, especially unless great care is taken in specifying the initial "start" value.

The canned-in default starting value, or "seed," ($x_0 = 5^{15}$) had been selected because theoretical results (Ref. 2) claimed that the longest nonrepeating sequences are obtained when x_0 is an odd power of 5. Unfortunately, the particular x_0 chosen, while perhaps generating a long sequence, does not seem to produce numbers with very good randomness properties.

This article describes an alternate generator of a type whose randomness has been theoretically shown to be vastly superior and which can be implemented on any computer, despite word length restrictions (the U1108 algorithm was tailored to 36-bit words). It is a machine-independent algorithm.

The generation method is almost as fast as the linear congruential method, but not quite. The ratio of speeds is about 3:1.

II. The Linear Recurrence Modulo 2 Method

In 1965, the author (Ref. 3) showed that numbers produced as successive binary words of length s taken L bits apart ($s \leq L$) from a linear (shift-register) bit-stream recursion of the form

$$a_n = c_1 a_{n-1} + \cdots + c_{p-1} a_{n-p+1} + a_{n-p} \quad \text{(modulo 2)}$$

form a pseudorandom sequence whenever the polynomial $f(x) = x^p + c_{p-1}x^{p-1} + \cdots + c_1 x + 1$ is primitive over $GF(2)$. The algebraic structure of these pseudorandom numbers provided a way of proving that, over randomly chosen starting values $a_0, \cdots, a_{p-1}$ in the numeric sequence, the correlation between numbers is essentially zero, being ostensibly equal to -2^{-p}, for all numbers in the sequence separated by less than $(2^p - s - 1)/L$, subject to the restriction $(L, 2^p - 1) = 1$. The author further showed that adjacent k-tuples of such numbers were uniformly distributed for $1 \leq k \leq (p/L)$.

The algorithm for computing the numbers is simple, especially when $f(x)$ is a primitive trinomial, say $x^p + x^q + 1$, where $q < p/2$. An even greater simplification is possible, as is shown in this article, when p is an even multiple of L, as is the case for the trinomial $x^{532} + x^{37} + 1$ (from Zierler and Brillhard (Ref. 4). The generator based on this polynomial has period 1.4×10^{160}, has virtually no (average) correlation between any numbers separated by less than 5×10^{158}, has 28-bit precision numbers available, and has adjacencies up to 19 dimensions uniformly distributed. Runs-up and -down statistics up to length 16 are impeccable. The period and maximum correlation distance are, in fact, so great that the generator would have to produce numbers at a nanosecond rate for more than 10^{142} years before nonrandom distribution or correlation effects would be noticeable as nonrandom. Almost 4×10^{14} numbers would have to be examined to detect deviations in runs-up and runs-down statistics as nonrandom.

These pseudorandom number generators have been widely studied (Refs. 5–8) since 1965, both theoretically and empirically, and have been "promoted to pride of place in the field of pseudorandom number generation (Ref. 7)."

Tootill (Ref. 8) has even discovered generators of this type for which "there can exist no purely empirical tests of the sequence as it stands capable of distinguishing between it and [truly random sequences]." For reasons having to do with computer storage and precision, the generator of this article is, unfortunately, not one of these. Nevertheless, the generator described is vastly superior to any linear congruential generator in existence.

III. The Method

The algorithm for producing the succeeding p bits from the current set of p bits in the $x^p + x^q + 1$ generator is,

(1) Left-shift the p bit string by q bits, inserting q zeros on the right, dropping q bits on the left.

(2) Add modulo 2 (exclusive-or) the original and shifted p bits.

(3) Right-shift this result by $p - q$ bits, supplying $p - q$ zeros on the left, and dropping $p - q$ bits on the right.

(4) Add the results of 2 and (3) above modulo 2 to give the next p bits.

The proof that this algorithm works is very simple, and the reader is invited to apply the algorithm to the bit string $a_1 a_2 \cdots a_p$ and use the reduction formula $a_{p+m} = a_{q+m} + a_m$ (try it with $a_1 a_2 a_3 a_4 a_5$ with $a_{5+m} = a_{2+m} + a_m$ to see what is happening).

Note in the method that the number of computations required for generating the next p bits grows at most linearly in p. Assuming $p >> L$, then partitioning the p generator bits into words of length L produces a number of words that also increases in proportion to p. Therefore, the number of computations for L bit precision random numbers, to first-order effects, is independent of the recursion degree p. Making p large, however, has advantages in increasing randomness properties.

It is true that as p increases, more and more registers are required to compute each new set of p bits, and shifting many registers at once presents a small inconvenience in most computer languages. These factors place a small speed and storage overhead on the generation process; but as we shall see, even this is not extreme due to the particular trinomial chosen.

The trinomial $x^{532} + x^{37} + 1$ ($p = 532$, $q = 37$) has several things to recommend it: (1) 532 is factorable into $28 \cdot 19$, which means that 19 words each having 28 bits precision can be generated all at once by the algorithm; (2) up to and including 19-tuples of adjacently produced random numbers will be uniformly distributed; (3) 28 and 19 are both relatively prime to the period ($2^{532} - 1$), so no ill effects occur as a result of beginning new words every 28 bits; (4) the period and correlation distance are so great as never to be witnessed in the lifetime of the universe; and (5) the $q = 37$ value produces good runs-up and runs-down statistics (Ref. 8), up to runs of length 16.

On converting 28-bit fixed-point mantissas to real numbers, 8-digit precision results. Some machines, such as the 1108, may have to reduce this precision in order to fit the floating-point exponent field into the word (the U1108 has only 27-bit mantissa precision). In such cases, the most significant bits of the generated words may always be retained without losing randomness qualities, so that all implementations produce essentially identical results, within machine precision. This philosophy is present in the algorithm which follows in Section IV.

The RANDOMIZE function which initializes the generator uses a multiplicative linear congruential algorithm to generate the first 19-number "seed," from which the rest of the generated numbers grow. The particular value for the multiplicator a in the algorithm

$$x_{n+1} \equiv a x_n \quad (\text{modulo } 2^L)$$

was chosen as $41{,}475{,}557_{10}$ for the $L = 28$ case from theoretical results published in Ahrens and Dieter (Ref. 9), who show that good randomness results when

$$a \equiv 5 \quad (\text{modulo } 8)$$
$$a \approx 2^{L-2}\xi$$

where $\xi = (5^{1/2} - 1)/2$ (the "golden section number").

IV. The Algorithm

Managing the 532-bit shift-register is the main trick in implementing the method of the previous section. The algorithm given in this section utilizes an array W_i, $i = 0, \cdots, 18$ of b-bit computer words ($b \geq 28$) sufficient to encompass the $p = 532$ span of bits to be operated on (and retained), and delivered in 28-bit chunks whenever the RANDOM function is invoked.

On machines having word sizes smaller than 28 bits, double words will have to be used for each word in the algorithm below. Two values of the RANDOMIZE argument that round to the same internal fixed-point representation will produce the same random sequence; conversely, every unique fixed-point representation of the argument produces a unique random sequence. In addition, so long as the value of the argument is the same and stays within the precisions of two differing machines, the sequences produced on each will be the same, within machine precision.

Counting the number of elemental operations (load, store, shift, etc.) for the algorithm, one finds about $10 + f$ operations per number generated, where f is the number of operations in "floating" the fixed-point number. The linear congruential algorithm requires only about $3 + f$ such operations, so the ratio of speeds is approximately 3:1.

The RANDOM function is about $23 + f$ operations long, as compared to $3 + f$ for its linear congruential form, and data storage is 21 words vs 2. However, even though the program requires perhaps 7 times as much core as the linear congruential form, the total is still probably under 50 locations, of negligible concern in most installations. The asymptotically random sequence of Tootill (Ref. 8) requires 607 words to store the array W_i alone. (This, coupled with the fact that only 23-bit precision was available in that generator, is why it was not considered here.)

The two functions, described in CRISP-PDL syntax (Ref. 10), are as follows:

```
Function:  RANDOMIZE(starter:universal real)
         <* This function declares and initializes a 19-element
         <* array, $W_0, \ldots, W_{18}$ with random numbers generated
         <* by a linear congruential method. An integer $I$ is
         <* set to zero to enable RANDOM to select $W_0$ as the
         <* first random number. $W$ and $I$ are permanent data
         <* structures, accessed only by RANDOMIZE and RANDOM
```

```
.1     constant multiplier:integer = 41475557
.2     variable j:integer, I:integer,
         W: array [0..18] of universal integer
.3     if    (starter<0) < * MBASIC manual specifications * >
.4     :      starter: =clocktime < * returns current time of day as integer * >
.5     :- >(starter=0)
.6     :      starter: =multiplier
       :- >(else) < * starter>0 * >
.7     :      starter: =fix(starter)< * floating-to-integer conversion * >
       :..endif
.8     $W_0$: =starter
.9     loop for j=1 to 18
.10    ↑ $W_j$ = ($W_{j-1}$ * multiplier) modulo 2 ** 28
.11    ←←repeat with next j
.12    I=0 < * set up to pick $W_0$ as first random number * >
       endfunction
```

Function: RANDOM:**real**

```
       < *  This algorithm makes use of a 19-word array,
       < *  $W_0, \ldots, W_{18}$, each with $b \geq 28$ bits. Each word contains
       < *  28 bits of the generator, right justified. An
       < *  integer variable I on entry contains the index of
       < *  the word array next to be returned as the random value.
       < *  Both W and I are permanent data structures, accessed
       < *  only by RANDOM and RANDOMIZE. The latter of these
       < *  initializes I to zero and W to the seed.

.1     variable j:integer
.2     if (I=19) < * all words have been used up * >
.3     :  I: =0 < * reset to first element in array * >
.4     :  loop for j=0 to 16 < * exclude last two words * >
.5     :  ↑ load $W_{j+1}, W_{j+2}$ into $A_0, A_1$
.6     :  ↑ left shift $A_1$ by $b-28$ < * join bits in stream * >
.7     :  ↑ left shift $A_0, A_1$ by 9 < * $q=37$ is $28+9$ * >
.8     :  ↑ $W_j$: =($W_j$ XOR $A_0$) < * the recursion formula * >
.9     :  ←←repeat with next j
.10    :  load $W_{18}, W_0$ into $A_0, A_1$ < * now compute $W_{17}$: * >
.11    :  left shift $A_1$ by $b-28$ < * join $W_{18}$, $W_0$ bit streams * >
.12    :  left shift $A_0, A_1$ by 9 < * $A_0$ now has final 19 bits of $W_{18}$ * >
.13    :  $W_{17}$: =$W_{17}$ XOR $A_0$ < * and first 9 bits of stream shifted 495 * >
.14    :  load $W_0, W_1$ into $A_0, A_1$ < * do similarly for $W_{18}$ * >
.15    :  left shift $A_1$ by $b-28$
.16    :  left shift $A_0, A_1$ by 9
.17    :  $W_{18}$: =($W_{18}$ XOR $A_0$)
       :..endif
.18    RANDOM =float($W_I$)/2 ** 28 < * convert to real * >
.19    I: =I+1
       endfunction
```

V. Conclusion

The algorithm given has been implemented as the RANDOM function in the MBASIC processor currently on the Caltech PDP-10 computer. All tests so far run on it validates the randomness properties claimed by the theory. In that theory, by the way, the only factor left to chance is the specification of the initial "seed." The stated uniformity, zero-correlation, and runs statistics are all based on the single assumption that the seed be chosen randomly. Of course, the default value canned in was not randomly chosen, but chosen specifically to look random except for the first word, and certainly, to the extent of our tests, this appears to have worked beautifully.

We have demonstrated that the generator is also capable (as is every known random number generator) of producing numbers with 3σ variations from randomness over a few thousand samples when the wrong seed is supplied.

References

1. MBASIC, Vol. I — Fundamentals, Jet Propulsion Laboratory, Pasadena, Calif., p. 188.
2. Knuth, D. E., *The Art of Computer Programming,* Vol. 2 — Seminumerical Algorithms, Addison-Wesley Co., Reading, Mass., 1969.
3. Tausworthe, Robert C., "Random Numbers Generated by Linear Recurrence Modulo Two," *Math Comp,* Vol. XIX, No. 90, Apr., 1965, pp. 201–208.
4. Zierler, N., and Brillhart, J., "On Primitive Trinomials (Mod 2), II," *Inform. Contr.,* Vol. 14, No. 6., June 1969, pp. 566–569.
5. Whittlesey, J. R. B., "A Comparison of the Correlational Behavior of Random Number Generators for the IBM 360," *Comm. ACM,* Vol. 11, No. 9, Sept., 1968, pp. 641–644.
6. Neuman, F., and Martin, C. F., "The Autocorrelation Structure of Tausworthe Pseudorandom Number Generators," *IEEE Trans. Comp.,* Vol. C-25, No. 5, May 1976, pp. 460–464.
7. Tootill, J. P. R., et. al., "The Runs-Up and -Down Performance of Tausworthe Pseudorandom Number Generators," *ACM J.,* Vol. 18, No. 3, July, 1971, pp. 381–399.
8. Tootill, J. P. R., et al., "An Asymptotically Random Tausworthe Sequence," *ACM J.,* Vol. 20, No. 3, July 1973, pp. 469–481.
9. Ahrens, J. H., et al., "Pseudorandom Numbers: A New Proposal for the Choice of Multiplicators," *Computing,* No. 6, Springer-Verlag, 1970, pp. 121–138.
10. Tausworthe, R. C., *Standardized Development of Computer Software,* SP 43-29, Jet Propulsion Laboratory, Pasadena, Calif., July 1976.