Version: 25.11.92 (To appear in ACM TOMS)

A Portable Uniform Random Number Generator Well Suited for the Rejection Method

W. Hörmann and G. Derflinger
University of Economics and Business Administration
Department of Statistics, Augasse 2-6, A-1090 Vienna, Austria

Abstract: Up to now all known efficient portable implementations of linear congruential random number generators with modulus $2^{31}-1$ are working only with multipliers which are small compared with the modulus. We show that for non-uniform distributions, the rejection method may generate random numbers of bad quality if combined with a linear congruential generator with small multiplier. Therefore a method is described that works for any multiplier smaller than 2^{30}. It uses the decomposition of multiplier and seed in high order and low order bits to compute the upper and the lower half of the product. The sum of the two halfs gives the product of multiplier and seed modulo $2^{31}-1$. Coded in ANSI-C and FORTRAN77 the method results in a portable implementation of the linear congruential generator that is as fast or faster than other portable methods.

CR Categories and Subject Descriptors: G.3 [Probability and Statistics]: Random number generation
General Terms: Algorithms
Additional Key Words and Phrases: Uniform random number generator, linear congruential generator, portability, rejection method, quality of non-uniform random numbers

1. Introduction

An old but still very popular method to generate uniform pseudo-random numbers on a computer is the linear congruential generator (LCG):

$$x_{i+1} = (a \cdot x_i + c) \bmod m$$

where the multiplier a, the increment c and the modulus m are positive integers which must be properly chosen (cf. [5]). In this paper we are only interested in the case that $c = 0$ and $m = 2^{31} - 1$. The resulting generator (it has a period of $2^{31} - 2$ if a is a primitive element modulo m) is widely discussed in literature as it allows efficient implementations on computers with a wordsize of 32 bits ([10], [5]). Recently portable implementations of the generator were discussed which are only valid for comparatively small multipliers a ([11], [2]). In [8] a FORTRAN implementation is suggested that applies the method of [11] twice. It is valid for all a that can be factorized into two integers smaller than 2^{15} but has the disadvantage of being rather slow. In [6] and [9] computer searches were performed to find the best multipliers (with respect to the lattice structure from dimensions two to six) for which the method in [2] is applicable. Compared with the best multipliers of [3] not much is lost by this restriction as far as the lattice structure is concerned.
Uniform random numbers are often used to generate non-uniform random numbers. There are many papers dealing with so-called exact methods for generating non-uniform random numbers provided that there is an ideal source of uniform random numbers. There are few papers dealing with the quality of non-uniform random numbers that are generated by transformation methods in combination with LCGs. In [1] the computation of the one-dimensional discrepancy indicates that the quality of the rejection method using inversion to sample from the dominating density

is bad when combined with a LCG with a small multiplier. In section 2 this observation is examined. Section 3 gives portable implementations for LCGs with $m = 2^{31} - 1$ and $a < 2^{30}$.

2. The Quality of the Rejection Method

The rejection method, introduced in 1951 by John von Neumann, is a flexible method to generate non-uniform random numbers. To sample from a distribution with density $f(x)$ we need a dominating density (or hat function) $h(x)$ and a real number α such that $\alpha f(x) \leq h(x)$ for all x. Now generate a random number X from the dominating density h and a random number V uniformly distributed on $(0, h(X))$. If $V \leq \alpha f(x)$ accept X as a random number with density f, otherwise reject X and try again. Figure 1 shows the rejection method for the beta(2,3) density with $f(x) = 12x(1-x)^2$ and $h(x) = 1$. The acceptance probabilty α is 0.5625. The points of Figure 1 are all pairs of the LCG with $m = 2^{10}$, $a = 33$ (which is close to $\sqrt{m}$) and $c = 1$. As it is well known (cf. eg. [5]) these points form a lattice. As a is in the order of $\sqrt{m}$ a short lattice vector is almost parallel to the y-axis. Therefore as one moves along the x-axis, alternately, several successive points are accepted and then several successive points are rejected. This results in relatively large intervals without an accepted point, which can be seen in Figure 1 where the projections of the accepted points are marked on the x-axis. It is obvious that the accepted points do not approximate the beta(2,3) distribution very well.

Figure 1

Easy considerations show that this problem occurs not only for $h(x)$ constant but for any rejection method that uses inversion of a probability distribution function to sample from $h(x)$ with $\alpha f(x)/h(x) \ll 1$. For the normal distribution and $h(x)$ the density of the Cauchy distribution see Figure 2. The points are all pairs returned by the LCG with $m = 2^9$, $a = 21$ (which is close to $\sqrt{m}$) and $c = 1$ as transformed by the rejection method.

Figure 2

As a measure for the quality of the approximation of a one-dimensional distribution we use the one-dimensional discrepancy of the sequence y_i with respect to the distribution function F of the desired distribution.

$$D_N^1(F) := \sup_{-\infty \leq s \leq t \leq \infty} \left| \frac{\#\{y_i : s < y_i \leq t, i = 1, \cdots, N\}}{N} - (F(t) - F(s)) \right|$$

We computed the discrepancy D_Z^1 of all non-uniform random numbers that can be returned by a rejection method combined with a certain LCG for the beta(2,3) distribution and $h(x) = 1$ and for the normal distribution with $h(x)$ the density of the Cauchy distribution. (For the results it is essential that the inversion method is applied to sample from the dominating distribution.) In addition we generated $n = 10^6$ samples from each distribution, and then we computed the χ^2-statistic using 10^5 subintervals of equal probability, starting the LCG with seed 1. The results of these computations are contained in Table 1.

Table 1

	$m \cdot D_Z^1$		χ^2-value	
Multiplier	beta	normal	beta	normal
742938285 suggested in [3]	164.27	193.79	99404.0	100015.8
950706376 suggested in [3]	233.14	188.81	100071.0	100195.2
630360016 suggested in [8]	148.34	202.38	99494.8	99529.0
397204094 used in SAS and IMSL	271.21	439.69	99894.6	100371.4
16807 the "minimal standard" of [9]	56799.39	48582.05	367131.6	206056.0
39373 found in [6]	24297.35	20789.49	214153.6	144163.4
48271 suggested in [9]	20076.95	16936.76	177831.2	131313.8
69621 suggested in [9]	13722.25	11733.92	131067.6	113625.0

The results of Table 1 support the considerations of above: The rejection method, using inversion to sample from the dominating density in combination with a LCG with a multiplier about as small as $\sqrt{m}$, produces random numbers with a bad one-dimensional distribution. Strictly speaking the one-dimensional resolution of the random numbers is low compared with the resolution of the LCG itself. That this fact can influence the results of a simulation experiment is shown by the χ^2-values as the critical value for $\alpha = 0.1$ is 100573 for $\alpha = 10^{-10}$ it is 102870.
As the rejection method in combination with inversion (or with a constant dominating density) is a widely used and flexible method for generating non-uniform random numbers the results of Table 1 justify the recommendation not to use LCGs with small multiplier. As these multipliers were mainly suggested because a LCG with small multiplier can be coded in a efficient and portable way (cf. [2]) we introduce an efficient portable implementation that is valid for $m = 2^{31} - 1$ and $a < m/2$ in the following section.

3. A Portable Implementation

To code $aX \bmod m$, $m = 2^{31} - 1$, we use the idea first described in [10], the integer part is denoted by $[.]$.

$$aX \bmod m = (aX \bmod (m+1) + [aX/(m+1)](m+1)) \bmod m =$$

$$= (aX \bmod (m+1) + [aX/(m+1)](m+1) - [aX/(m+1)]m) \bmod m =$$

$$= (aX \bmod (m+1) + [aX/(m+1)]) \bmod m$$

To compute $aX \bmod 2^{31} + [aX/2^{31}]$ in a high level language it is necessary to divide a and X into high order and low order bits. Combining these two techniques we get Algorithm 1 that computes $aX \bmod (2^{31} - 1)$ for $a < 2^{30}$ which is enough for all practical purposes.

Algorithm 1

0. Constants $ahi = [a/2^{15}]$,
   $alo = a \bmod (2^{15})$.

1. Set $xhi \leftarrow [x/2^{16}]$,
   $xlo \leftarrow x \bmod (2^{16})$,
   $mid \leftarrow ahi \cdot xlo + 2\,alo \cdot xhi$.

2. Set $x \leftarrow ahi \cdot xhi + [mid/2^{16}] + alo \cdot xlo$.
   If $x > 2^{31} - 1$ set $x \leftarrow x - (2^{31} - 1)$.

3. Set $x \leftarrow x + 2^{15}(mid \bmod 2^{16})$.
   If $x > 2^{31} - 1$ set $x \leftarrow x - (2^{31} - 1)$.

As $a < 2^{30}$ ahi, alo and xhi are less than 2^{15} and xlo is less than 2^{16}. Therefore it is obvious that Algorithm 1 only uses integers between 0 and $2^{32} - 1$. As the type unsigned long int of the ANSI-C standard must have at least 32 bits we can give the following ANSI-C code of a linear congruential generator with $m = 2^{31} - 1$ and $a < 2^{30}$.

```
unsigned long int x=1;
#define A 742938285
#define AHI (A>>15)
#define ALO (A&0x7FFF)
double rand()
{
   unsigned long int xhi,xlo,mid;
   xhi=x>>16;
   xlo=x&0xFFFF;
   mid=AHI*xlo+(ALO<<1)*xhi;
   x=AHI*xhi+(mid>>16)+ALO*xlo;
   if (x&0x80000000) x-=0x7FFFFFFF;
   x+=((mid&0xFFFF)<<15);
   if (x&0x80000000) x-=0x7FFFFFFF;
   return (x*(1./2147483647.0));
}
```

For computer languages that have no unsigned integers we give the following variant of Algorithm 1 to compute $aX \bmod (2^{31}-1)$ for $a < 2^{30}$. Algorithm 2 works correctly on any computer that can handle integers between $-(2^{31}-1)$ and $(2^{31}-1)$.

Algorithm 2

0. Constants $ahi = [a/2^{15}]$,
   $alo = a \bmod (2^{15})$.
1. Set $xhi \leftarrow [x/2^{16}]$,
   $xlo \leftarrow x \bmod (2^{16})$,
   $mid \leftarrow (-(2^{31}-1) + ahi \cdot xlo) + 2\, alo \cdot xhi$.
   If $mid < 0$ set $mid \leftarrow mid + (2^{31}-1)$.
2. Set $x \leftarrow (-(2^{31}-1) + ahi \cdot xhi) + [mid/2^{16}] + alo \cdot xlo$.
   If $x > 0$ set $x \leftarrow x - (2^{31}-1)$.
3. Set $x \leftarrow x + 2^{15}(mid \bmod 2^{16})$.
   If $x < 0$ set $x \leftarrow x + (2^{31}-1)$.

Algorithm 2 (FORTRAN77-implementation):

```
      DOUBLE PRECISION FUNCTION RAND(X)
      IMPLICIT INTEGER (A-Q,S-Z)
      DOUBLE PRECISION R
      PARAMETER (T=2**15,U=2**16,M=2**30-1+2**30,W=-M,R=1.D0/M,
     * A=742938285,AHI=A/T,ALO=A-T*AHI,ALO2=2*ALO)
      XHI=X/U
      XLO=MOD(X,U)
      MID=W+AHI*XLO+ALO2*XHI
      IF(MID.LT.0) MID=MID+M
      X=W+AHI*XHI+MID/U+ALO*XLO
      IF(X.GT.0) X=X+W
      X=X+T*MOD(MID,U)
      IF(X.LT.0) X=X+M
      RAND=X*R
      END
```

We tested the C-version of Algorithm 1 with different compilers and computers (DEC-station 5200 + Ultrix Compiler, PC 386/25 with Interactive Unix C-compiler and PC 386/20 with Turbo-C) and it was as fast or faster than the portable method for small multipliers suggested in [2] and recommended in [6] and [9]. The FORTRAN version of Algorithm 2 was tested on our DEC-station. Depending on the optimization flag it is between 1.2 and 2 times slower than the C-program. Of course the idea of Algorithms 1 and 2 can be used to implement multiple recursive or matrix linear congruential generators (cf. [4] or [7]) in an efficient and portable way as well.

References

[1] Afflerbach, L., and Hörmann, W. Nonuniform random numbers: a sensitivity analysis for transformation methods. in: International Workshop on Computationally Intensive Methods in Simulation and Optimization, U. Dieter and G. Ch. Pflug, ed., Lecture Notes in Econom. Math. Systems 374, 135-144, Springer-Verlag, Berlin, (1992).

[2] P. Bratley, P., Fox, B. L., and Schrage, L. E. A Guide to Simulation. Springer-Verlag, New York, (1983).

[3] Fishmann, G. S., and Moore, L. R. An exhaustive analysis of multiplicative congruential random number generators with modulus $2^{31} - 1$. SIAM J. Sci. Stat. Comput. 7, 1 (Jan. 1986), 24-45.

[4] Grothe, H. Matrix generators for pseudo-random vector generation. Statistical Papers 28 (1987), 233-238.

[5] Knuth, D. E. The Art of Computer Programming. Vol. II, Addison-Wesley, Menlo Park, London, Amsterdam, Don Mills, Sydney, (1981).

[6] L'Ecuyer, P. Efficient and portable combined random number generators. Communications of the ACM 31, 6 (June 1988), 742-749.

[7] L'Ecuyer, P. Random numbers for simulation. Communications of the ACM 33, 10 (Oct. 1990), 85-97.

[8] Marse, K., and Roberts, S. D. Implementing a portable FORTRAN uniform (0,1) generator. Simulation 41, 4 (Oct. 1983), 135-139.

[9] Park, S. K., and Miller, K. W. Random number generators: good ones are hard to find. Communications of the ACM 31, 10 (Oct. 1988), 1192-1201.

[10] Payne, W. H., Rabung, J. R., and Bogyo T. P. Coding the Lehmer pseudo-random number generator. Communications of the ACM 12, 2 (Feb. 1969), 85-86.

[11] Schrage, L. A more portable FORTRAN random number generator. ACM Transactions on Mathematical Software 5, 2 (June 1979), 132-138.