Computer Physics Communications 221 (2017) 299–303

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

A revision of the subtract-with-borrow random number generators☆

Alexei Sibidanov

University of Victoria, Victoria, BC, Canada V8W 3P6

ARTICLE INFO

Article history:
Received 30 May 2017
Received in revised form 17 August 2017
Accepted 4 September 2017
Available online 11 September 2017

Keywords:
Linear congruential generator
Subtract-with-borrow generator
RANLUX
GMP

ABSTRACT

The most popular and widely used subtract-with-borrow generator, also known as RANLUX, is reimplemented as a linear congruential generator using large integer arithmetic with the modulus size of 576 bits. Modern computers, as well as the specific structure of the modulus inferred from RANLUX, allow for the development of a fast modular multiplication — the core of the procedure. This was previously believed to be slow and have too high cost in terms of computing resources. Our tests show a significant gain in generation speed which is comparable with other fast, high quality random number generators. An additional feature is the fast skipping of generator states leading to a seeding scheme which guarantees the uniqueness of random number sequences.

Program summary/New version program summary
Program Title: RANLUX++
Licensing provisions: GPLv3
Programming language: C++, C, Assembler

1. Introduction

The well known Linear Congruential Generator (LCG) is a recurrent sequence of numbers calculated as follows:

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

where x_0 is the initial state or *seed*, a — the multiplier, c — the increment and m — the modulus. The particular choice of the parameters a, c and m with period q — the minimal number when $x_q = x_0$, can be found in the literature [1]. Commonly used LCGs are limited to $m \leq 2^{64}$, and have poor statistical properties. Thus they are not used for Monte-Carlo physical simulations.

This situation can be mitigated when m reaches several hundreds or even thousand bits. The cost of the increased range of m is to deal with arbitrary precision integer arithmetic which was believed to be prohibitively expensive for practical purposes. In the last two decades there has been tremendous progress in modern central processor units (CPU) especially for personal computers (PC) which can be employed for long arithmetic.

We have explored the possibility to use the long arithmetic in LCG to improve the quality of generated random numbers and found that, despite a substantial increase in calculations, the time to generate a single random number is not proportionally risen. In fact for some parameters, the computational time decreased compared to ordinary LCGs with machine word modulus size.

2. Subtract-with-borrow generator

At this point no specific constraints on a, c and m parameters of LCG have been applied. As a good starting point we choose the subtract-with-borrow generator first introduced in [2] and the intimate connection with LCG has been shown as a part of the period calculation. The algorithm has been extensively studied in [3] to improve statistical quality of generated numbers. Based on this study the generator RANLUX [4] was developed and now it is widely used in physics simulations as well as in other fields where random numbers with high statistical quality are required. However the current method employed by RANLUX to achieve the high quality makes it one of the slowest generators on the market.

The definition of the subtract-with-borrow generator is the following: let b be some integer greater than 1 also called *the base* and vector $Y = (y_1, \ldots, y_r, k)$ with the length $r + 1$, where $0 \leq y_i < b$ and k or *the carry* equals 0 or 1. Then define a recursive transformation of the vector Y_i with the rule:

$$Y_{i+1} = \begin{cases} (y_2, \ldots, y_r, \Delta, 0), & \text{if } \Delta \geq 0 \\ (y_2, \ldots, y_r, \Delta + b, 1), & \text{otherwise} \end{cases} \tag{2}$$

where $\Delta = y_{r-s+1} - y_1 - k$ and r and s also called *the lags*. As shown in the work [5], this recursion is equivalent to LCG with the modulus $m = b^r - b^s + 1$, the multiplier $a = m - (m - 1)/b$ and

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

E-mail address: sibid@uvic.ca.

http://dx.doi.org/10.1016/j.cpc.2017.09.005
0010-4655/

Algorithm 1 Calculating remainder using only additions, subtractions and bit shifts for the modulus $m = b^{24} - b^{10} + 1$.

```
1: procedure REMAINDER(z)                          ▷ 0 ≤ z < b^48
2:     t0 ← [z0, ..., z23]                         ▷ 0 ≤ t0 < b^24
3:     t1 ← [z24, ..., z47]                        ▷ 0 ≤ t1 < b^24
4:     t2 ← [z38, ..., z47]                        ▷ 0 ≤ t2 < b^10
5:     t3 ← [z24, ..., z37]                        ▷ 0 ≤ t3 < b^14
6:     r ← t0 − (t1 + t2) + (t3 + t2) · b^10
7:     c ← ⌊r/b^24⌋                    ▷ floor function rounds to −∞
8:     r ← r − c · m                               ▷ 0 < r < b^24
9:     return r
10: end procedure
```

$c = 0$ with the relation:

$$x_i = x(Y_i) \equiv \sum_{j=1}^{r} y_j b^{j-1} - \sum_{j=1}^{s} y_{r-s+j} b^{j-1} + k. \tag{3}$$

A reverse transformation to get a corresponding sequence of the subtract-with-borrow generator requires to calculate the digits of the fractional expansion in base b of x_i/m:

$$\begin{aligned} x_i/m &= 0.y_r y_{r-1} y_{r-2} \ldots y_1 y_0 y_{-1} \ldots, \\ k_0 &= I[(y_0 - y_s + y_r + 1) \bmod b = 0], \\ k &= I[y_s - y_0 - k_0 < 0], \end{aligned} \tag{4}$$

where the function I returns 1 if the condition in the brackets is true and 0 otherwise.

In the RANLUX generator the lags $r = 24$ and $s = 10$ with the base $b = 2^{24}$ are chosen among other suggested parameters in [2], and thus the modulus $m = b^{24} - b^{10} + 1$ is a prime number and the multiplier $a = m - (m-1)/b = b^{24} - b^{23} - b^{10} + b^9 + 1$. With those parameters the period q is equal to $(m-1)/48$.

Due to the selected base b the natural choice to keep the generator state is a vector of length 24 composed of 24-bit numbers. This implementation uses the properties of the modulus m to avoid long arithmetic calculations, and a single step equivalent to one modular multiplication $(a \cdot x \bmod m)$ that requires only subtraction of two 24-bit numbers and carry propagation. In the original FORTRAN implementation, 24-bit numbers were stored as floats to avoid at that time, a high cost integer-to-float conversion.

2.1. Remainder

The simple structure of the modulus m allows us to calculate the remainder using only additions, subtractions and bit shifts. The modulus m and thus the generator state x have size of $24 \cdot 24 = 576$ bits and fits into 9 64-bit machine words. The result of the product $z = a \cdot x$ fits into 18 64-bit machine words which can be represented as a 48 element array of 24-bit numbers: $z = [z_0, z_1, \ldots, z_{46}, z_{47}]$. The number r obtained by the procedure shown in Algorithm 1 is congruent to $z \bmod m$ and $r < b^{24}$. Note the product $c \cdot m$ is also only bit shifting due to the simple structure of m. The calculation of c is a sum of carry bits of each arithmetic operation.

2.2. Skipping

Examining the result of a single step of Eq. (1) one can note that the main part of the number x_i is preserved in its successor x_{i+1} which is just rotated by 24 bits. This strong correlation is the reason of the poor statistical quality of the original subtract-with-borrow generator [2]. The bright idea developed in [3] is to apply the transformation (2) many times to break correlations between nearby states before using the state for actual physical simulation. The drawback of this method is obvious – all intermediate states have to be explicitly calculated even if they are not needed. Despite the single step being simple with small resource consumption, good statistical quality requires several hundred steps thus in total, the skipping requires a lot of time. This is a luxury to spend resources and not use the results. Thus so-called luxury levels were introduced as aliases for how many generated numbers have to be wasted.

Using Eq. (1) we can efficiently skip numbers since all p recurrent steps collapse to a single multiplication:

$$\begin{aligned} \underbrace{a \cdot (a \cdot (\ldots) \bmod m) \bmod m}_{p \text{ times}} &= (a^p \bmod m) \cdot x \bmod m \\ &= A \cdot x \bmod m, \end{aligned} \tag{5}$$

where the factor $A \equiv (a^p \bmod m)$ is precomputed and thus the cost to calculate the next state with or without skipping is the same. Any state in the entire period $q = (m-1)/48 \approx 10^{171}$ can be calculated in no more than $2 \times \log_2(q) \approx 1140$ long multiplications using fast exponentiation by squaring which takes order of tens of μs on modern CPUs.

In Table 1 the precomputed values of $A \equiv (a^p \bmod m)$ where the values of p is taken from [4] are shown for illustrative purposes. In the initial rows, long chains of 0 or 1 in binary representation are clearly visible and this can be interpreted such that for each bit of the state x_{i+p} only a few bits of the state x_i contributes. Even at the highest luxury level 4 there are still some patterns observable and a demanding user maybe not be completely satisfied. For such user the two last rows would be more attractive especially since it is for free! Such chaotic multipliers mean that if any single bit of the state x_i is changed in the next step the altered state will be absolutely different from the unaltered one.

With explicit long multiplication, there is no need to keep the multiplier A as a power of a, it can be adjusted to get the full period, $m-1$. As an example the number $(a^{2048} + 13 \bmod m)$ is a primitive root modulo m and with this multiplier all numbers in the range $1 \ldots m-1$ will appear in the sequence only once with any initial x_0 from the same range.

The fast skipping also provides a seeding scheme which guarantees non-colliding sequences of random numbers — what is needed is to just skip a large enough sequence. This is suggested in [5] using the seed number as a power of the multiplier:

$$x_s = (a^n)^s \cdot x_0 \bmod m, \tag{6}$$

where s is the seed, x_0 is the initial state which is the same for all seeds, x_s is the starting state to deliver random numbers for the seed s and n is the seed skipping factor to guarantee non-colliding sequencies for any Monte Carlo simulation. Lets take $n = 2^{96} \approx 10^{29}$ which is so big that for any modern computer the required time to visit all states within the range exceeds the age of the Universe, even if it can generate a new state on every CPU clock cycle. In this case the maximum seed number is $s < q/n \approx 2^{474} \approx 10^{143}$.

3. Implementation

The core procedure for efficient implementation of LCG is fast long multiplication. This provides the infrastructure to deliver random numbers in an efficient as well as convenient form to users. Programming languages such as C++, C or FORTRAN which are usually used in Monte Carlo physics simulations, have no native support of the long arithmetic despite all desktop grade CPUs providing hardware instructions suitable for it. The support of long arithmetic is provided by external libraries. We choose the GMP library [6] for proof of concept. This library is highly optimized for basic arithmetic operations with long numbers and supports a large number of CPU architectures. It was found that GMP is a great