Implementarea unui generator de numere pseudo-aleatorii

Un generator bun de numere pseudo-aleatorii este caracterizat de câteva criterii importante aplicate valorilor produse: uniformitate, independență, eficiență și imprevizibilitate. Pentru implementarea unui astfel de generator pot fi utilizați mai mulți algoritmi: LCG (linear congruential generator), LFG (lagged Fibbonacci generator), LFSR (linear feedback shift registers), permutări unidirecționale, etc.

Linear Congruential Generator

Basic Introduction

A linear congruential generator represents an algorithm that produces a sequence of pseudo-randomized numbers determined with a discontinuous piecewise linear equation. The method is well known for being one of the oldest and most spreaded pseudorandom number generators algorithm. The logic behind it is very easy to be understood as well as easy and fast to be implemented, especially if the computer hardware can provide modular arithmetic by storage-bit truncation.

The generator uses as its main core the recurrence relation:

**Xn+1 = (a \* Xn + c) % m**

Where X represents the sequence of pseudorandom values, and

m, 0 < m – the modulus a, 0 < a < m – the multiplier c, 0 <= c < m – the increment X0, 0 <= X0 < m – the seed or start value

Are integer constants that describe the generator. For example, if c=0, the generator would be called a multiplicative congruential generator(MCG), or Lehmer RNG. On the other hard, if c != 0, the method is called a mixed congruential generator.

In terms of math, when c != 0 the recurrence relation would represent rather an affine transformation than a linear one, but computer science has established better the inaccurate name.

Period Length

A benefical thing about LCGs is that depending on the choice of its parameters’ values, the period is known and also more important long. Although it is not the only criterion that affects a pseudorandom number generator being flawed, a period being too short can be clearly fatal.

Generally, LCGs can produce pseudorandom numbers that can pass the formal tests for randomness, but the quality of the output it generates is very responsive to the choice of parameters m and a. Let’s take as an example, a = 1 and c = 1 that produce a simple modulo-m counter, which has a long period, but clearly isn’t as random as we would want it to be.

Mainly, there exists the three common groups of parameter choice for LCGs :

1. **M prime, c = 0**

This is also known as the Lehmer RNG construction. The period is m-1 only if the multiplier a is picked as a primitive element of the integers modulo m. In the initial state the first value must be chosen between 1 and m-1.

One disavantage of a prime modulus is the fact that the modular lowering needs a double-width product and an explicit reduction step. Usually a prime that is less than a power of 2 is prefered(popular choices are 231−1 and 261−1), so that we have the formula for the reduction modulo *m* = 2*e* − *d* calculatedinsuchawayas(*ax* mod 2*e*) + *d* ⌊*ax*/2*e*⌋. Although, this needs a follow-up with a conditional substraction of m if the result is too big, but the number of substractions is limited by the rule ad/m, which can be easily transformed into one if d is small.

As a fact we have the situation in which if a double-width product is unreasonable, and the multiplier is chosen precisely, Schrage’s method may be used. To do this, use the factor m = q \* a + r, i.e. q =[m/a] and r = m mod a. Then compute a \* x mod m = a \* (x mod q) - r \* [x / q]. Since we have x mod q < q <= m/a, the first term is strictly less than a \*m / a =m. If a is chosen in order to let r <= q, then the second term is also less than m: r \* [x / q] <= r \* x / q = x \* (r / q) <= x < m. Thus, both products can be calculated in the end with a single-width product and the difference between them lies in the sequence [1-m, m-1], so it can be transformed to [0, m-1] with a single condition add.

Another disadvantage is that is uncommon to convert the value 1 <= x < m to uniform random bits. If a prime that is less than a power of 2 have been used, the missing values should be simply ignored.

1. **M a power of 2, c = 0**

M being a power of 2, usually *m* = 232 or *m* = 264, results in the creation of an efficient LCG, because this allows the modulus operation to be computed using the simple shortening of the binary representation. Actually, the most significant bits aren’t computed at all. However, disadvantages are still bound to appear.

This form has the maximum period of m / 4, achieved if a ≡ 3 or a ≡ 5. The initial state *X*0 must be an odd number, and the lowest three bits of X be alternated between two states and they are not useful. It can be shown that this form is as good as a generator with a modulus a quarter the size and c != 0. A more important problem that comes with the usage of a power-of-two modulus is the fact that the low bits have a shorter period than the high bits. The lowest-order bit of X never changes(X is always odd), and the next two bits oscilate between two states. The rule follows like this: if *a* ≡ 5 (mod 8), then bit 1 never changes and bit 2 alternates. If *a* ≡ 3 (mod 8), then bit 2 never changes and bit 1 alternates. Bit 3 repeats with a period of 4, bit 4 has a period of 8, and so on. Only the most significant bit attains the full period.

1. **c != 0**

If c != 0, chosen parameters can influence a period equal to m to happen no matter the seed values. Although this will occur only if these conditions are respected:

* m and c are prime,
* a - 1 is divisible by all prime factors of m,
* a - 1 is divisible by 4 if m is divisible by 4

This form may work with any m, but it is bound to succeed only if m has many repeated prime factors such as powers of 2. If m was a square-free integer, this would allow *a* ≡ 1, which makes a very bad PRNG; a selection of full-period multipliers is available when m has repeated prime factors.

Even tho, this theorem produces a maximum period, it is not enough to guarantee a good generator. As an example, it is desirable for a – 1 to not be any more divisible by prime factors of m than necessary. Thus, if m is a power of 2, then a – 1 should be divisible by 4 but not divisible by 8 to respect the rule, i.e. a ≡ 5.

Note that a power-of-2 modulus has the same problem as described above for c = 0: the low k bits form a generator with modulus 2*k* and thus repeat with a period of 2*k*; only the most significant bit achiving the full period.

The generator doesn’t care about the choice of c, as long as it is relatively prime to the modulus(the simplest example would be m a power of 2 resulting in c being odd), so the value c = 1 is commonly chosen.

The series created by other picks of c can be illustrated as a simple function of series when c = 1. Getting deeper into it, if Y is the test series defined by *Y*0 = 0 and *Yn*+1 = *a \* Yn* + 1 mod m, then a general series *Xn*+1 = *a \* Xn* + *c* mod *m* can be written as an affine function of *Y*:

*Xn* = (X0 \* (a – 1) + c) \* *Yn* + X0 =(X1 - X0) \* *Yn* + X0 (mod m).  
A more usual illustration would be that any two series X and Z with the same multiplier and modulus are related according to the relation:

*(Xn* - X0) / (X1 - X0) = Yn = (a*n* – 1) / (a – 1) = (Zn - Z0) / (Z1 - Z0)

PseudoCode

Begin

Declare class mRND

Create a function Seed(number)

Assign a variable \_seed=number

Create a constructor mRND

Declare \_seed(0), a(0), c(0), m(2147483648)

Create a function rnd()

Return

\_seed = (a \* \_seed + c) mod m

Declare a, c, m, \_seed

Done

Declare an another subclass MS\_RND inheriting from base class mRND

Create a constructor

Read the variables a, c

Create a function rnd()

return mRND::rnd() right shift 16

Done

Declare an another subclass SS\_RND inheriting from base class mRND

Create a constructor

Read the variables a, c

Create a function rnd()

return mRND::rnd()

Done

For x=0 to 6

Print MS\_RAND

For x=0 to 6

Print SS\_RAND

Done

End

The Good And Bad Parts of using LCGs

LCGs are fast and use minimal memory in order to keep their composure. This makes them worth of simulating multiple independent streams.

Mainly, there are some specific disavantages of using LCGs, the most problematic one would be the fact that the state is too small. People who used them for so many years can prove the fact that the technique is so good in spite of the issues. A LCG with big enough state can even pass stringent statistical tests.

For a specific example, an ideal random number generator with 32 bits of output is most likely to begin reproducing earlier values after √*m* ≈ 216 results. Any pseudo random number generator which has its output its full, untruncated state will not create duplicates untill the full period happens. For the fact, PRNGs should have a period longer than the number of outputs required. According to the speed of modern computers, a period of 264 would be good for all but a lower one for the least demanding applications, and longer ones for the simulations.

A specific flaw to LCGs is that if the dimesional space is of n-dimension, the points will lie on *n*√*n*!⋅*m*. This is happenning because of the connection between the successive values in the sequence Xn. If a(the multiplier) is chosen carelessly, the planes will be fewer and wider and most likely will conduct problems. The spectral test, which is a simple test of an LCG’s quality, measures this spacing and can lead to a good multiplier being chosen.

Another flaw mainly found in LCGs is the short period of the low-order bits when is m is a power of 2. This can be avoided by using a modulus larger than the required output, and also using the most significant bits of the state.

Although, LCGs can be a good option in some situations. As an example, in an embedded system, the memory is severly capped. Another situation can be found in a video game console taking a small number of high-order bits of an LCG may well be enough. The low order bits go through very short cycles.

LCGs should be evaluated with a lot of attention for fitting in non-cryptographic applications where highly advanced randomness is critical. In Monte Carlo simulations, a LCG must use a modulus greater than the cube of the number of random samples which are needed. This means, a good 32-bit LCG can be suited for obtaining one thousand random numbers; a 64-bit LCG can be good for approximatively 2 milion random numbers. Taking this in consideration, LCGs are not the best for Monte Carlo simulations.