Random Number Generation

CMS 380 Simulation and Stochastic Modeling

Pseudorandom Number Generators

High-quality random numbers are the building blocks of simulation programs. Every modern programming environment includes a method for generating, at minimum, a uniformly distributed random value in the range $[0, 1)$.

First, consider the problem of generating a "true" random number on a computer. If the "random" value is being generated by a program, it can't *really* be truly random, because there must be some underlying deterministic code calculating the value. Therefore, programs that calculate streams of random numbers are better termed *pseudorandom number generators* (PRNGs). A good PRNG is one that can produce long streams of values that have the same statistical properties as true random values.

It is possible to generate "true" random values if you sample from a random physical process. Various services have been created that do things like sample from the atmospheric background radiation, which is effectively random. It is also possible to generate random values by sampling from human user input.

The Linear Congruential Generator

A *linear congruential pseudorandom number generator* has the form

$$X_{n+1} = aX_n + c \mod m$$

It generates a stream of values starting from an initial starting value X_0, which is called the *seed*.

Here's an example of the sequence produced when $a = 5$, $c = 1$, $m = 16$, and $X_0 = 0$.

$$\begin{aligned}
X_0 &= & \text{seed} &= 0 \\
X_1 &= & (5 \cdot 0 + 1) \mod 16 &= 1 \\
X_2 &= & (5 \cdot 1 + 1) \mod 16 &= 6 \\
X_3 &= & (5 \cdot 5 + 1) \mod 16 &= 15 \\
X_4 &= & (5 \cdot 15 + 1) \mod 16 &= 12 \\
X_5 &= & (5 \cdot 12 + 1) \mod 16 &= 13 \\
X_6 &= & (5 \cdot 13 + 1) \mod 16 &= 2 \\
X_7 &= & (5 \cdot 2 + 1) \mod 16 &= 11 \\
X_8 &= & (5 \cdot 11 + 1) \mod 16 &= 8 \\
X_9 &= & (5 \cdot 8 + 1) \mod 16 &= 9 \\
X_{10} &= & (5 \cdot 9 + 1) \mod 16 &= 14 \\
X_{11} &= & (5 \cdot 14 + 1) \mod 16 &= 7 \\
X_{12} &= & (5 \cdot 7 + 1) \mod 16 &= 4 \\
X_{13} &= & (5 \cdot 4 + 1) \mod 16 &= 5 \\
X_{14} &= & (5 \cdot 5 + 1) \mod 16 &= 10 \\
X_{15} &= & (5 \cdot 10 + 1) \mod 16 &= 3 \\
X_{16} &= & (5 \cdot 3 + 1) \mod 16 &= 0
\end{aligned}$$

$X_{16} = X_0$, so the sequence would repeat from that point forward.

Period. An LCG can only produce values in the range 0 to $m - 1$, because that's the output range of the mod operation. The generator above has *full period*: it cycles through all 16 possible values for $m = 16$ before repeating. Having a long period is a good property for a PRNG, because you can then use it in a simulation to generate long sequences of random values without worrying about repetition.

Having a large m is necessary, but the relationship between a, c, and m is also important and complex. There are theorems that explain the relationships that must exist between the three parameters for the generator to achieve full period; you can read more about them if you're interested.

Parameters. Choosing the values of a, c, and m is a challenging problem. It requires careful testing and validation to find a stream that produces both a long period and a stream of random numbers with good randomness properties. For example, Java's `Random` class uses $m = 2^{48}$, $a = 25214903917$, and $c = 11$.

Because parameterizing a PRNG is hard, **you should never write your own to use in production code** unless you have no other choice. It's too easy to come up with an implementation that seems superficially correct, but ends up being biased or inaccurate in some way.

There was, in fact, a famous bug in a C function called `RANDU`, which had the formula

$$X_{n+1} = 65539 \cdot X_n \mod 2^{31}$$

This turns out to produce random number streams that are highly correlated, meaning that there is a dependence between successive values, which makes them definitely not random enough for valid simulation results. In fact, if the initial seed is odd (which it was in implementations), the formula would only generate odd numbers! When the `RANDU` bug was finally discovered, it required re-validation of some simulation results that could no longer be trusted due to the low quality of its random number streams.

Uniform Random Variables. One last point: the LCG generates integers in $[0, m-1]$, but we often want floating point values in $(0, 1)$. If the generator has a good stream and m is large, then:

$$U_n = \frac{X_n}{m}$$

is a good approximation of a uniformly distributed random variable in $(0, 1)$.