A Recap of Randomness

Random number generating algorithms are rated by two primary measures: entropy - the measure of disorder in the numbers created and period - how long it takes before the PRNG begins to inevitably cycle pattern. While high entropy and a long period are desirable traits, it is sometimes necessary to settle for a less intense method of random number generation to not sacrifice performance of the product the PRNG is required for. However, in the real world PRNGs must also be evaluated for memory footprint, CPU requirements, and speed.

In this poster we will explore three of the major types of PRNGs, their history, their inner workings, and their uses.

The Mersenne Twister

The Mersenne Twister is the most widely used general purpose pseudorandom number generator today. A Mersenne prime is a prime number that is one less than a power of two, and in the case of a mersenne twister, is its chosen period length (most commonly $2^{19937} - 1$). There are only 50 such numbers known to exist today, with the most recent, $2^{77,232,917} - 1$, only being found on January 3rd of this year. Makoto Matsumoto and Takuji Nishimura developed the algorithm in 1997 to overcome the flaws found in older PRNGs, being the first PRNG to provide high entropy, long period random number generation in little time. Because of this, the Mersenne Twister is the default pseudorandom number generator for software systems such as Microsoft Excel, GAUSS, GNU, IDL, MATLAB, Python, R, and Ruby.

There are many reasons to choose the mersenne twister over other PRNGs. As stated, it has a truly impressive speed and quality combination, producing even 64-bit floats 20x faster than hardware based solutions, while also passing statistical tests for randomness like TestU01's BigCrush suite, however this is true for all PRNGs based on linear recurrence, including the Mersenne Twister. It has also been referred to as being a mixed congruential generator. Removing c creates a formula used for a pure multiplicative generator: $X_n = aX_{n-1} \mod m$

Notice that after four numbers have been generated in the previous example, the sequence begins to repeat. This is due to a modular arithmetic that forces wrapping of values into the desired range resulting in the period of 4. The value that primarily affects the period of the sequence is m, where larger m's result in a much longer period.

Unfortunately, a long period does not guarantee a random sequence. A sequence made from this formula can have sufficient randomness but a very short period, or it can have a long period with an obviously non random sequence. The most common way to work around this problem is to remove c from the equation by making it 0. With c, the formula can be referred to as being a mixed congrential generator. Removing c creates a formula used for a pure multiplicative generator: $X_n = aX_{n-1} \mod m$

This results in a simpler algorithm, so many random number algorithms use the second equation over the first. An important note about the pure multiplicative generator formula is that $aX_{n-1}$ cannot generate 0, or every subsequent value will become 0. This is not an issue with mixed congruential generators (c is not equal to 0) because if $aX_{n-1}$ results in zero, the constant c would be added to it.

A major advantage of Linear Congruential Generators is that they require minimal memory and therefore are generally faster than other PRNGs. They are ideal for when an application's requirements for strong randomness are not essential. Being faster than other PRNGs, however, makes it less efficient than others. It is highly recommended to use a different more random generator for high level applications. LCGs are much better suited for smaller level applications, such as an environment like a video game console.

In conclusion, LCGs better fit smaller applications where speed is of the utmost importance.