Generating Random Numbers and Random Variables

At the heart of Monte Carlo simulation is a sequence of pseudo-random numbers generated to drive the simulation. It is "pseudo" because the random numbers are in fact produced by a completely deterministic algorithm. Our objective is to present a simple uniform random number generator (algorithm) that are good enough for all practical purposes. We then describe a general method for transforming uniform random numbers into random samples with specified probability distributions and an algorithm based on Central Limit theorem to generate standard normal random variable.

Before discussing sequences that appear to be random but are not, we should specify what we mean by a generator of genuinely random numbers. We mean a mechanism of producing a sequence of random variables \( U_1, U_2, \ldots \) with the properties:

1. Each \( U_i \) are uniformly distributed between 0 and 1;
2. The \( U_i \) are mutually independent.

Property (1) is convenient but arbitrary normalization; Values uniformly distributed between 0 and 1/2 would be just as useful, as they would be from any other simple distribution. Uniform random variables on \([0,1]\) can be transformed into samples from essentially any other distribution using methods discussed below. Property (2) is the more important one. It implies that all pairs of rv should be independent and, in particular, the value of \( U_i \) should not be predicted from \( U_{i-1}, \ldots, U_1 \).

A pseudorandom number generator produces a finite sequence of numbers in the unit interval. The values obtained depend, in part, on the input parameters specified by the user. Any such sequence constitutes a set of possible outcomes of independent uniforms. An effective generator produces values that appear consistent with properties (1) and (2). If the number of values \( K \) is large, the fraction of values falling in any subinterval of the unit interval should be approximately the length of the subinterval—this is uniformity. Independence suggests that there should be no discernible pattern among the values.

Uniform Random Numbers Generator

A congruential generator is a recursive formula returning a sequence of pseudo-random numbers. It starts with an initial (seed) value \( x_0 \). Recall that for any positive integer \( m \), the modulus-\( m \) of \( x \), denoted by \( x \pmod{m} \), is defined as the remainder of \( x \) after division by \( m \). In other words,

\[
x \pmod{m} = x - \left\lfloor \frac{x}{m} \right\rfloor m
\]

where \( \lfloor x \rfloor \) denotes the greatest integer less than or equal to \( x \). Our focus is on the following generator.

**Algorithm 1 (Linear congruential generator)**

1. Fix positive integers \( m \) (modulus) and \( a \) (multiplier).
2. Set up a seed \( x_0 \in \{1, 2, \ldots, m-1\} \).
3. Run the recursive formula \( x_{i+1} = ax_i \pmod{m} \).
4. Return \( u_{i+1} = \frac{x_{i+1}}{m} \in [0, 1] \).

The main feature of a linear congruential generator is that the random numbers \( u_1, u_2, \ldots \) generated will eventually repeat themselves. It is of course of best interest to find generators that produce more distinct values before repeating. The following table displays moduli and multipliers for seven linear congruential generators that have been recommended in the literature. The period for each of the generators below is its modulus \( m \).

<table>
<thead>
<tr>
<th>Modulus ( m )</th>
<th>Multiplier ( a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 2^{24} - 1 = 2,147,483,647 )</td>
<td>39,373</td>
</tr>
<tr>
<td>( 2,147,483,399 )</td>
<td>40,692</td>
</tr>
<tr>
<td>( 2,147,483,563 )</td>
<td>40,014</td>
</tr>
</tbody>
</table>

(2)
A fast implementation of the algorithm is possible using only integer arithmetic, while avoiding overflow (nonexact representation of the product $ax_i$). The implementation is based on the following observation.

Let $q = \lceil \frac{m}{a} \rceil$ and $r = m \mod a$. By (1) the modulus can be represented as $m = aq + r$. The calculation to be carried out by the generator is

$$ax_i \mod m = ax_i - \left[ \frac{ax_i}{m} \right] m$$

$$= \left( ax_i - \left\{ \frac{x_i}{q} \right\} m \right) + \left( \left\{ \frac{x_i}{q} \right\} - \left\{ \frac{ax_i}{m} \right\} \right) m$$

$$= \left( ax_i - \left\{ \frac{x_i}{q} \right\} \left( aq + r \right) \right) + \left( \left\{ \frac{x_i}{q} \right\} - \left\{ \frac{ax_i}{m} \right\} \right) m$$

$$= \left( a \left( x_i - \left\{ \frac{x_i}{q} \right\} q \right) - \left\{ \frac{x_i}{q} \right\} r \right) + \left( \left\{ \frac{x_i}{q} \right\} - \left\{ \frac{ax_i}{m} \right\} \right) m$$

To prevent overflow, we need to avoid calculation of the potentially large term $ax_i$. It turns out that if $a \leq \sqrt{m}$ then $\left( \left\{ \frac{x_i}{q} \right\} - \left\{ \frac{ax_i}{m} \right\} \right)$ takes on values 0 or 1. The condition $a \leq \sqrt{m}$ is satisfied by the moduli and multipliers in table (2). Also, if $\left\{ \frac{x_i}{q} \right\} - \left\{ \frac{ax_i}{m} \right\} = 0$ then $a \left( x_i - \left\{ \frac{x_i}{q} \right\} q \right) - \left\{ \frac{x_i}{q} \right\} r \in \{0, 1, ..., m - 1\}$, and if $\left\{ \frac{x_i}{q} \right\} - \left\{ \frac{ax_i}{m} \right\} = 1$ then $a \left( x_i - \left\{ \frac{x_i}{q} \right\} q \right) - \left\{ \frac{x_i}{q} \right\} r \in \{-m + 1, -m + 2, ..., -1\}$. Hence, we can implement the calculation of $ax_i \mod m$ by checking indirectly whether $\left\{ \frac{x_i}{q} \right\} - \left\{ \frac{ax_i}{m} \right\}$ results 0 or 1. This ensures every intermediate calculation results in an integer between $-m + 1$ and $m - 1$. The following gives an implementation of this idea.

**Algorithm 2 (Linear congruential generator)**

1. Fix positive integers $m$ (modulus) and $a$ (multiplier) such that $a \leq \sqrt{m}$.
2. Compute the integer constants $q = \lceil \frac{m}{a} \rceil$ and $r = m \mod a$.
3. Set up a seed $x_0 \in \{1, 2, ..., m - 1\}$.
4. for $i = 0$ up to $N$ do
5. $k = \left\lfloor \frac{x_i}{q} \right\rfloor$
6. $x_{i+1} = a \left( x_i - kq \right) - kr$
7. If $(x_{i+1} < 0)$ then $x_{i+1} = x_{i+1} + m$
8. Return $u_{i+1} = \frac{x_{i+1}}{m} \in [0, 1]$.
9. end do

**Inverse Transform Method**

This is the simplest method for simulating a random variable with a given cumulative distribution function (c.d.f.) $F$. Given a uniform random variable $U$ on $[0, 1]$, we look for a transformation $f$ of $U$ such that $f(U)$ has c.d.f. given by $F$, that is

$$P(f(U) \leq x) = F(x).$$

(3)

If $f$ is bijective and monotonically increasing, the inverse function $f^{-1}$ is well defined and we may write

$$P(f(U) \leq x) = P(U \leq f^{-1}(x))$$

$$= f^{-1}(x)$$

(4)

Comparing expressions (3) and (4) suggests that any function $f$ whose inverse $f^{-1}$ matches $F$ is the required transformation. We consider three cases on how we should define $f$.

**Case 1:** $F$ is continuous and increasing. Then $F$ is bijective and we define $f = F^{-1}$. 

2
Case 2: $F$ is continuous and nondecreasing (increasing but not one-to-one). We define $f(y) = \min \{ x | F(x) = y \}$.

Case 3: $F$ is discontinuous. $F(x) = \sum_{i : x_i \leq x} p_i$ where $p_i$ is the probability of $x_i$. We define $f(y) = \min \{ x | F(x) \geq y \}$.

Algorithm 3 (Inverse transformation)

1. Simulate $U \sim \text{Unif}[0, 1]$.
2. Return $f(U)$.

Example 1. Exponential Distribution.
The exponential distribution with mean $\theta$ has distribution $F(x) = 1 - e^{-\frac{x}{\theta}}, x \geq 0$. Inverting the exponential distribution yields the algorithm $X = -\theta \log (1 - U)$. This can also be implemented as $X = -\theta \log (U)$ because $U$ and $1 - U$ have the same distribution.

Example 2. Arcsine Law.
The time at which a standard Brownian motion attains its maximum over the time interval $[0, 1]$ has distribution $F(x) = \frac{2}{\pi} \arcsin (\sqrt{x}), 0 \leq x \leq 1$.

The inverse transform method for sampling from this distribution is $X = \sin^2 \left( \frac{U \pi}{2} \right), U \sim \text{Unif}[0, 1]$. Using the identity $2 \sin^2 (t) = 1 - \cos (2t)$ for $0 \leq t \leq \frac{\pi}{2}$, we can simplify the transformation to $X = \frac{1}{2} - \frac{1}{2} \cos(U\pi), U \sim \text{Unif}[0, 1]$.

Example 3. Discrete Distribution.
Consider a discrete random variable whose possible values are $c_1 < c_2 < \ldots < c_n$. Let $p_i$ be the probability attached to $c_i, i = 1, \ldots, n$. Set $q_0 = 0$ and

$$q_i = \sum_{j=1}^{i} p_j, i = 1, \ldots, n.$$ 

These are the cumulative probabilities associated with the $c_i$, that is, $q_i = F(c_i), i = 1, \ldots, n$. To sample from this distribution,

i) Generate a uniform random variable $U$.

ii) Find $K \in \{1, \ldots, n\}$ such that $q_{K-1} < U < q_K$.

iii) Set $X = c_K$.

The second step can be implemented through binary search.

Simulate Univariate Standard Normal Random Variable

The central limit theorem states that the ratio

$$\frac{\sum_{i=1}^{n} X_i - n\mu}{\sqrt{n\sigma}}$$

asymptotically converges to the standard normal $N(0, 1)$. Set $X_i = U_i \sim \text{Unif}[0, 1]$. Then $\mu = 1/2, \sigma = 1/\sqrt{12}$ and for $n$ sufficiently large, the ratio

$$\frac{\sum_{i=1}^{n} X_i - \frac{n}{2}}{\sqrt{\frac{n}{12}}}$$

is approximately normal. The assignment $n = 12$ provides a sufficiently accurate approximation for most simulation purposes.
Algorithm 4 (Normal distribution by summing up uniforms)

1. Generate $U_1, \ldots, U_{12} \sim Unif[0, 1]$.

2. Return $X = \left( \sum_{i=1}^{12} U_i \right) - 6$.

Let us conclude with a few remarks. For a log-normal random variable, we may return $e^X$, where $X$ is as described in the algorithm above. We cannot use the inverse transformation method to generate a normal sample because the c.d.f. admits no analytical expression. Analytical approximations to this c.d.f. actually exist, but inverting them involves a root searching algorithm that may make the resulting sampling procedure slower.