1 Introduction to Monte Carlo Simulation

The core idea of Monte Carlo is to understand a system by simulating with the help of random sampling. Let us explain the idea using the following example.

Example 1. Let we want to **estimate the average distance** between two randomly selected points in a region. This is important in many practical setting. For example, the energy needed by a bird to maintain it's territory depends on the average distance between it's nest and points of the area. In wireless networks, the connectivity depends on average distance between different components. Such problem can be meaningful if we try to find average distance between hospital or fire station and houses in a locality.

Let $\boldsymbol{X} = (X_1, X_2)$ and $\boldsymbol{Y} = (Y_1, Y_2)$ be independent and uniformly distributed two points drawn from a finite rectangle $R = [0, a] \times [0, b]$. The Euclidean distance between these two points is

$$Z = d(\boldsymbol{X}, \boldsymbol{Y}) = \sqrt{(X_1 - Y_1)^2 + (X_2 - Y_2)^2}.$$

We can **approximate** $E(Z)$ by sampling pair points $(\boldsymbol{X}_i, \boldsymbol{Y}_i)$, $i = 1, 2, \ldots, n$, from R and then calculating the average

$$\frac{1}{n} \sum_{i=1}^{n} d(\boldsymbol{X}_i, \boldsymbol{Y}_i). \qquad ||$$

Thus, in a simple Monte Carlo problem, we **express the quantity we want to know as the expected value of a random variable** Y, such as $\mu = E(Y)$. Then we generate values $Y_1, \ldots, Y_n$ independently from the distribution of Y and take their average

$$\widehat{\mu}_n = \frac{1}{n} \sum_{i=1}^{n} Y_i$$

as an estimate of μ. This process is quite intuitive.

Remark 1. In many examples, $Y = f(\boldsymbol{X})$, where the random vector $\boldsymbol{X}$ has a probability density function $p(\boldsymbol{x})$, and f is a real-valued function defined over the support of $\boldsymbol{X}$. Then $\mu = \int f(\underline{x}) p(\underline{x}) d\underline{x}$. In still other settings, $\boldsymbol{X}$ is a discrete random variable with a probability mass function that we also call p. †

2 Justification for Simple Monte Carlo

Let Y be a random variable for which $\mu = E(Y)$ exists, and suppose that $Y_1, \ldots, Y_n$ are independent and identically distributed random variables with the same distribution as Y. Then, using the weak law of large numbers (WLLNs),

$$\lim_{n \to \infty} \mathbb{P}\left(|\widehat{\mu}_n - \mu| \leq \varepsilon\right) = 1,$$

holds for any $\varepsilon > 0$. The WLLN tells us that our **chance of missing by more than ϵ goes to zero**. The strong law of large numbers (SLLNs) tells us a bit more. The **absolute error $|\widehat{\mu}_n - \mu|$ will eventually get below ϵ** and then **stay there forever**:

$$\mathbb{P}\left(\lim_{n\to\infty} |\widehat{\mu}_n - \mu| = 0\right) = 1.$$

Loosely speaking, the SLLNs says that the error in approximation will be very small if we increase n. Note that even if the error is small, there remains some error. We will discuss about the measurement of error as we progress.

Thus, to use Monte Carlo simulation, we **need to generate random number from appropriate distribution**. In the **first part** of the course, we will talk about different procedure of generation of random numbers. In the **second part** of the course, we will talk about different process of reducing error in Monte Carlo estimates.

3 Generation of Random Numbers

First we will discuss the procedure to **generate random number from uniform distribution**. As we proceed, we will see that random numbers can be generated from **several other distributions** based on random numbers from uniform distribution. Thus, the algorithms to generate random number from uniform distribution are the **core** of Monte Carlo simulation methods. These algorithms may be executed millions of times in the course of a simulation, making efficient implementation especially important.

3.1 Random Numbers Vs Pseudo Random Number

It would be better if we can generate random number based on a process, which is **truly random** based on well established knowledge. One such process could be **radioactive particle** emission, that are thought to be truly random. However, there are several drawback to use such a process. For example, it is very **difficult to rerun the process** to generate purely random numbers. Normally, we need thousands of random number for a mathematical computation. Therefore, we need to **save** all the random numbers. Therefore, **in practice, pseudo random numbers are used**. Pseudo random number generator **try to mimic the randomness and do not generate random numbers that are truly random**.

3.2 Random Number Generation (General Consideration)

As discussed, at the heart of nearly all Monte Carlo simulations is a **sequence of apparently random numbers** used to drive the simulation. We will treat this driving sequence as though it was genuinely random. Modern *pseudorandom* number generators are **sufficiently good at mimicking genuine randomness** (even though they are **produced by completely deterministic algorithm**). 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 aim at a mechanism for producing a sequence of random variables $U_1, U_2, \ldots$ with the property that:

1. Each U_i is **uniformly distributed** between 0 and 1.
2. The U_i are **mutually independent**.

Note:

1. Property 1 is convenient but can be arbitrarily normalized. This means that values uniformly distributed between 0 and 1/2 would be just as useful. Uniform random variables on the unit interval can be transformed into samples from essentially any other distribution.

2. Property 2 is more important. In other words, all pairs of values should be uncorrelated and, more generally, that the values of U_i should not be predictable from $U_1, U_2, \ldots, U_{i-1}$.

A random number generator (often called the *pseudorandom* number generator to emphasize that it only mimics randomness) produces a finite sequence of numbers $u_1, u_2, \ldots, u_K$, in the unit interval. An effective generator therefore produces values that appear consistent with properties 1 and 2. If the number of values of K is large, the **fraction of values falling in any subinterval should be approximately the length of the subinterval** – this is **uniformity**. **Independence** suggests that there **should be no discernible pattern among the values**. More precisely, **statistical tests for independence** would not easily reject segments of the sequence $u_1, u_2, \ldots, u_K$.

3.3 Linear Congruence Generator

A linear congruence generator is recurrence relation of the following form:

$$\begin{aligned} x_{i+1} &= ax_i \bmod m, \\ u_{i+1} &= x_{i+1}/m. \end{aligned}$$

Here the ***multiplier*** a and the ***modulus*** m are integer constants that determine the values generated, given an initial value (***seed***) x_0. The seed is an integer between 1 and $m-1$ and is ordinarily specified by the user. The operation $y \bmod m$ returns the **remainder** of y (an integer) after division by m. In other words $y \bmod m = y - \lfloor y/m \rfloor m$ where $\lfloor x \rfloor$ denoted the **greatest integer less than or equal** to x. For example, 7 mod 5 is 2 and 43 mod 5 is 3 . Because the result of mod m is always an integer between 0 and $m-1$ the output values u_i produced **are always between** 0 and $(m-1)/m$. In particular, they lie in the unit interval. Simplicity and potential effectiveness has lead to wide usage of linear congruence generators, in practice. Notice that the **linear congruence generator has the form** $x_{i+1} = f(x_i)$ and $u_{i+1} = g(x_{i+1})$ for some deterministic functions f and g.

Example 2. Suppose $a = 6$ and $m = 11$. If $x_0 = 1$ then we get the sequence

$$1, 6, 3, 7, 9, 10, 5, 8, 4, 2, 1, 6, \ldots$$

and the entire sequence repeats. All digits are included before a value was repeated. ||

Example 3. Suppose $a = 3$ and $m = 11$. If $x_0 = 1$ then we get the sequence $1, 3, 9, 5, 4, 1, \ldots$ and if $x_0 = 2$ then we get the sequence $2, 6, 7, 10, 8, 2, \ldots$. We see that regardless of the choice of x_0, $a = 3$ produces five distinct numbers before repeating, unlike the case of $a = 6$. ||

A linear congruence generator that produces all $(m-1)$ distinct values before repeating is said to have *full period*. Accordingly, we have the following general consideration in the construction of a random number generator.

1. **Period Length**: Any random number generator so generated will eventually repeat itself. Other things being equal, generators with longer periods are preferred. The longest possible period for a linear congruential generator with modulus m is $m-1$.

2. **Reproductivity**: One drawback of a genuine random sequence is that it can not be reproducible easily. Reproduciblity can be accomplished with a linear congruential generator by using the same seed x_0.

3. **Speed**: This is needed for random number generators since they may be called thousands or even millions of times in a single simulation.

4. **Portability**: An algorithm for generating random numbers should produce the same sequence of values on all computing platforms.

5. **Randomness**: Depends on (1) Theoretical Properties (2) Statistical Tests.

3.4 General Linear Congruence Generators

The general linear congruence generator (first proposed by Lehmer) takes the form:

$$\begin{aligned} x_{i+1} &= (ax_i + c) \bmod m \\ u_{i+1} &= x_{i+1}/m. \end{aligned}$$

This is sometimes called a ***mixed*** linear congruential generator and the multiplicative case in the previous section a ***pure*** linear congruential generator. Like a and m, the parameter c must also be an **integer**. Simple conditions that ensures that the generator has full period, that is, the number of distinct values generated from any seed x_0 is $(m-1)$ are listed below:

1. If $c \neq 0$ the conditions are (Knuth):

   (a) c and m are relatively prime (their only common divisor is 1).

   (b) Every prime that divides m divides $(a-1)$.

   (c) $(a-1)$ is divisible by 4 if m is.

   A simple consequence is that if m is a power of 2, the generator has a **full period** if c is odd and $a = 4n+1$ for some integer n.

2. If $c = 0$ and m is prime, **full period is achieved** from any $x_0 \neq 0$ if:

   (a) $a^{m-1} - 1$ is a multiple of m.

   (b) $a^j - 1$ is not a multiple of m for $j = 1, 2, \ldots, m-2$.

   A number a satisfying these two properties is called a ***primitive root*** of m. When $c = 0$, the sequence x_i becomes $\{x_0, ax_0, a^2x_0 \ldots\} (\text{mod } m)$ and the sequence returns to x_0 at smallest k for which $a^k x_0 \bmod m = x_0$, that is, the smallest k for which $a^k \bmod m = 1$.

3.5 Other Generators

Combining Generators:
One can move beyond the basic linear congruence generators by combining two or more such generators through summation. Wichmann and Hill proposed summing values in the unit interval. L'Ecuyer sums first and then divides. To be more explicit, consider J generators with the j-th generator having parameters a_j and m_j. Then,

$$\begin{aligned} x_{j,i+1} &= a_j x_{j,i} \bmod m_j, \\ u_{j,i+1} &= x_{j,i+1}/m_j \ , \ j = 1, 2, \ldots, J. \end{aligned}$$

1. The **Wichmann-Hill** combination sets u_{i+1} equal to the fractional part of $u_{1,i+1} + u_{2,i+1} + \cdots + u_{J,i+1}$.

2. **L'Ecuyer's** combination takes the form:

$$x_{i+1} = \sum_{j=1}^{J} (-1)^{(j-1)} x_{j,i+1} \bmod (m_1 - 1)$$

and

$$u_{i+1} = \begin{cases} x_{i+1}/m_1 \ , \ x_{i+1} > 0, \\ (m_1 - 1)/m_1 \ , \ x_{i+1} = 0. \end{cases}$$

This assumes that m_1 is the largest of the m_j.

A **combination of generators** can result in a **much longer period** than any of its components. A long period can also be achieved in a single generator by using a **larger modulus**, which could create problems with overflow. In combining generators, it is possible to choose the multiplier a_j much smaller than $\sqrt{m_j}$, in order to use the integer implementation. While L'Ecuyer's uses integer arithmetic, Wichmann-Hill uses floating point arithmetic. Another way of extending the basic linear congruence generator uses a higher order recursion of the form:

$$x_i = (a_1 x_{i-1} + a_2 x_{i-2} + \cdots + a_k x_{i-k}) \bmod m \text{ and } u_i = x_i/m.$$

This is called a ***multiple* recursive generator**. A seed for this generator consists of initial values $x_{k-1}, x_{k-2}, \ldots, x_0$. The vector $(x_{i-1}, x_{i-2}, \ldots, x_{i-k})$ can take up to m^k distinct values. The sequence x_i repeats once this vector returns to a previously visited value. If the vector reaches the zero vector then all subsequent x_i are zero. Thus the longest possible period for the multiple recursive generator is $m^k - 1$.

Inversive Congruential Generator:
This method uses recursions of the form

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

where the (mod m)-**inverse** x^- of x is an integer in $\{1, 2, \ldots, m-1\}$ satisfying $x x^- = 1 \bmod m$.

Fibonacci Generators:

The original Fibonacci recursion motivates the formula:

$$N_{i+1} = (N_i + N_{i-1}) \bmod M.$$

It turns out that this is **not suitable for generating** random numbers. The modified approach is:

$$N_{i+1} = (N_{i-\nu} - N_{i-\mu}) \bmod M,$$

for suitable $\nu, \mu \in \mathbb{N}$ is called the ***lagged Fibonacci*** generator. For many choices of ν and μ, this approach leads to recommendable generators.

Example 4. $U_i = U_{i-17} - U_{i-5}$. In case $U_i < 0$ we set $U_i = U_i + 1.0$. This recursion immediately produces floating point numbers $U_i \in [0, 1)$. This generator requires a prologue in which 17 initial U's are generated by means of another method.

||