Random Number Generators

Professor Karl Sigman
Columbia University
Department of IEOR
New York City
USA

Introduction

Your computer “generates" numbers $U_1, U_2, U_3, \ldots$ that are considered independent and *uniformly randomly distributed* on the continuous interval $(0, 1)$.

Recall that the probability distribution (cumulative distribution function) of such a uniformly distributed random variable U is given by

$$F(x) = P(U \leq x) = x, \ x \in (0, 1),$$

and more generally, for $0 \leq y < x < 1$,

$$P(y < U \leq x) = x - y.$$

In other words, the probability that U falls in an interval $(y, x]$ is just the length of the interval, $x - y$.

Introduction

Independence means that regardless of the values of the first n random numbers, $U_1, \ldots, U_n$, the value of the next one, U_{n+1}, still has the same uniform distribution over $(0, 1)$; it is not in any way effected by those previous values.

This is analogous to sequentially flipping a (fair) coin: regardless of the first n flips, the next one will still land heads (H) or tails (T) with probability $1/2$.

The sequence of random variables (rvs) $U_1, U_2, \ldots$ is an example of an *independent and identically distributed(iid) sequence*. Here, the identical distribution is the uniform over $(0, 1)$, which is a continuous analog of “equally likely" probabilities.

Introduction

In Python, for example, you can obtain such U as follows:

```
import random

U = random.random()
```

Once random is imported, then each time you use the command
U = random.random()
you receive a new uniform number within $[0, 1)$.

Introduction

It turns out once we have access to such uniform numbers U, we can use them to construct (“simulate/generate") random variables of any desired distribution, construct stochastic processes such as random walks, Markov chains, Poisson processes, renewal processes, Brownian motion and many other processes.
Suppose for example, that we want a random variable X that has an exponential distribution at rate λ: The cumulative distribution function (CDF) is given by

$$F(x) = P(X \leq x) = 1 - e^{-\lambda x},\ x \geq 0.$$

Introduction

Then simply define

$$X = -\frac{1}{\lambda}\ln(U),$$

where $\ln(y)$ denotes the natural logarithm of $y > 0$.
Proof:

$$\begin{aligned} P(X \le x) &= P(-\frac{1}{\lambda}\ln(U) \le x) \\ &= P(\ln(U) \ge -\lambda x) \\ &= P(U \ge e^{-\lambda x}) \\ &= 1 - e^{-\lambda x}. \end{aligned}$$

(Recall that $P(U \ge y) = 1 - y,\ y \in (0,1)$.)

Pseudorandom numbers

It turns out that the numbers generated by a computer are not really random nor independent as we said, but what are called *Pseudorandom numbers*.

This means that they appear, for all practical purposes, to be random (and independent) in the sense that they would pass various statistical tests for checking the random/independent property. We thus can use them in our simulations as if they were truly random—and we do.

Next we will discuss how the computer generates these numbers. This is deeply related to "cryptography" in which one wants to hide important information (from an opponent/enemy) in data that “appears" to be random.

Pseudorandom numbers

As an example to get you to think: Suppose I hand you a sequence of zeros and ones:

$$(0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1).$$

I tell you that I flipped a fair coin 25 times where $1 =$ Head (H), and $0 =$ Tails (T).

How can you check with certainty that I am telling the truth?

ANSWER: You can't

Pseudorandom numbers

But if I keep handing you such sequences of various lengths, then you can perform statistical tests that would help you decide if the sequences are consistent with coin flips.

Linear Congruential Generators

The most common and easy to understand and implement random number generator is called a *Linear Congruential Generator* (LCG) and is defined by a recursion as follows:

$$Z_{n+1} = (aZ_n + c) \ mod \ m, \ n \geq 0,$$

$$U_n = Z_n/m,$$

where $0 < a < m, \ 0 \leq c < m$ are constant integers, and *mod m* means *modulo m* which means you divide by m and leave the remainder. For example $6 \ mod \ 4 = 2$, $2 \ mod \ 4 = 2$, $7 \ mod \ 4 = 3$, $12 \ mod \ 4 = 0$. Thus all the Z_n fall between 0 and $m - 1$; the U_n are thus between 0 and 1.

$0 \leq Z_0 < c$ is called the *seed*. m is chosen to be very large, usually of the form $m = 2^{32}$ or $m = 2^{64}$ because your computer architecture is based on 32 or 64 bits per word; the modulo computation merely involves truncation by the computer, hence is immediate.

Linear Congruential Generators

For example if $m = 2^3 = 8$, then in binary, there are 8 numbers representing $\{0, 1, 2, 3, 4, 5, 6, 7\}$ given by a 3-tuple of 0s and 1s, denoted by

$$(i_0, i_1, i_2) = i_0 2^0 + i_1 2^1 + i_2 2^2, \ i_j \in \{0, 1\}, \ j = 0, 1, 2.$$

Thus, $(0, 0, 0) = 0$, $(1, 0, 0) = 2^0 = 1$, $(1, 1, 0) = 2^0 + 2^1 = 3$, $(0, 0, 1) = 2^2 = 4$ and $(1, 1, 1) = 2^0 + 2^1 + 2^2 = 7$, and so on.
Note how $7 + 1 = 8 = 0$ *mod* 8 is computed by truncation: $(1, 1, 1) + (1, 0, 0) = (0, 0, 0, 1) = 2^3$. The last component gets truncated yielding $(0, 0, 0) = 0$.
$10 = (0, 1, 0, 1)$ gets truncated to $(0, 1, 0) = 2$, and so on.

Linear Congruential Generators

Here is a more typical example:

$$Z_{n+1} = (1664525 \times Z_n + 1013904223)\ mod\ 2^{32}.$$

Thus $a = 1664525$ and $c = 1013904223$ and

$$m = 2^{32} = 4,294,967,296;$$

more than 4.2 billion.

Linear Congruential Generators

The numbers a, c, m must be carefully chosen to get a “good" random number generator, in particular we would want all c values $0, 1, \dots c - 1$ to be generated in which case we say that the LCG has *full period* of length c. Such generators will cyclically run thru the numbers over and over again.

To illustrate, consider

$$Z_{n+1} = (5Z_n + 1) \; mod \; 8, \; n \geq 0,$$

with $Z_0 = 0$. Then

$$(Z_0, Z_1, \dots, Z_7) = (0, 1, 6, 7, 4, 5, 2, 3),$$

and $Z_8 = 16 \; mod \; 8 = 0$, hence causing the sequence to repeat.

Linear Congruential Generators

If we increase c to $c = 16$,

$$Z_{n+1} = (5Z_n + 1) \; mod \; 16, \;\; n \geq 0,$$

with $Z_0 = 0$, then

$$(Z_0, Z_1, \ldots, Z_{15}) = (0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3),$$

and $Z_{16} = 16 \; mod \; 16 = 0$, hence causing the sequence to repeat.

Linear Congruential Generators

Choosing good numbers a, c, m involves the sophisticated use of number theory; prime numbers and such, and has been extensively researched/studied by computer scientists and mathematicians for many years.

Note that the numbers generated are entirely deterministic: If you know the values a, c, m, then once you know one value (Z_0, say) you know them all. But if you are handed a long sequence of the U_n, they certainly appear random, and that is the point.

Linear Congruential Generators

The advantages of using a LCG:

1. Very fast to implement
2. Requires no storage of the numbers, only the most recent value.
3. *Replication*: Using the same seed, you can generate exactly the same sequence again and again which is extremely useful when comparing alternative systems/models: By using the same numbers you are reducing the variability of differences that would be caused by using different numbers; any difference in the comparisons are thus due to the inherent difference in the models themselves.

More sophisticated generators

Python currently uses the Mersenne Twister as its core random number generator; U = random.random(). It produces at double precision (64 bit), 53-bit precision (floating), and has a period of $2^{19937} - 1$ (a Mersenne prime number). The Mersenne Twister is one of the most extensively tested random number generators in existence. (There is both a 32-bit and a 64-bit implementation.) It is not a LCG, it is far more complex, but yet again is deterministic and recursive. The basic Mersenne Twister algorithm continues to be refined and modified over the years to make it faster to implement, etc.