**Chapter 07: Random Number Generation**

Table of Contents

[Random Numbers 3](#_Toc87178120)

[Pseudorandom Values 5](#_Toc87178121)

[Considerations 6](#_Toc87178122)

[Midsquare Method 7](#_Toc87178123)

[Linear Congruential Generators 8](#_Toc87178124)

[Full Cycle Generators 9](#_Toc87178125)

[Choosing Parameters 10](#_Toc87178126)

[Multiplicative Generators 12](#_Toc87178127)

[Prime Modulus Multiplicative LCG 14](#_Toc87178128)

[More General Congruence 15](#_Toc87178129)

[Quadratic Congruential Generator 15](#_Toc87178130)

[Multiple Recursive Generator 15](#_Toc87178131)

In simulation programs, when providing input data, we frequently need to generate **random values**. There are actually two categories of random values:

1. Random Numbers
2. Random Variates

A **random number** is a number in between and , distributed **uniformly**.

A **random variate** is a randomly generated value that is **not a random number**. Thus, if we generate a random value from a distribution that is not uniformly distributed between and , the generated value is a random variate. We have already seen one of these, the randomly generated values from exponential distributions.

For now, we will assume that a random variate is magically found, but we will be looking into exactly how it works later on.

## Random Numbers

A sequence of numbers, , , are called **random numbers** when they possess two statistical properties:

1. Uniformity
2. Independence

**Uniformity** means that the numbers are uniformly distributed between and .

**Independence** means that the generation of one random number does not affect the probability of the generation of any of the random numbers.

Thus, each of the random numbers is an **independent sample** drawn from the **uniform distribution** .

The **PDF** is:

The **expected value** of the random numbers is:

The **variance** of the random numbers is:

Logically though, how do we ensure uniformity and independence? Say we divide the interval into classes of equal length. Uniformity and independence ensure that, if there are observations,

* The **expected number** of observations in each class is
* The **probability** of observing a value in a particular class is independent of the previous observations

## Pseudorandom Values

The random values we will be generating are not **truly random**. They are said to be **pseudorandom**. Essentially, this is because we generate them using mathematics, which inherently means they can be replicated, i.e. they are not truly random. In fact, it is not really possible to generate ‘truly’ random values.

Although this seems like a problem, using pseudorandom values in simulations can actually be beneficial. In simulations, we frequently want to **compare** performance between different systems. If we use truly random values, every system would get a different set of values, which means the comparisons would not be fair. Using pseudorandom values allows us to use the same set of randomly generated values for different systems and perform a fair comparison.

The negative bits of pseudo random values are:

* They may not be **uniformly distributed**
* They may be **discrete valued** instead of continuous
* The **mean** and **variance** may be too high or too low
* There may be **dependence** between the numbers in the form of:
  + **Autocorrelation**, where two subsets of the values from the same set of values are related to one another
  + Numbers getting successively **higher/lower**
  + Several numbers **above the mean** followed by several numbers **below the mean** or vice versa

### Considerations

There are a few considerations we need to make when generating pseudorandom numbers:

* The process needs to be **fast** and **memory efficient**, since we will be generating a huge set of numbers.
* The process needs to be **portable** to different computers and even programming languages.
* Since these are pseudorandom numbers, values will repeat. The set of values that are being repeated is called the **cycle**. We need to ensure that the cycle is **sufficiently large**, before the numbers begin to repeat.
* We should be able to **replicate** the random numbers if we wish to.
* We should be able to generate **separate streams**, essentially meaning that we should be able to parallelly generate multiple sets of values, each with their own set of criteria. This is done to minimize dependency between the streams.
* We need to be able to approximate the **statistical properties** of random numbers, i.e. uniformity, independence, maximum density (size of discrete set of values being used should be maximized to replicate continuity).

## Midsquare Method

The steps of generating a random number using the **Midsquare Method** are:

1. Start with a -digit positive integer, . This is the **seed** value.
2. Square to obtain an -digit number. Append s to the left if required.
3. Take the middle digits of as .
4. Place the decimal point to the left of to get the first random number, .
5. will be the middle digits of , and will be with a decimal point to the left.
6. Continue in this pattern.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

The problems with this method are:

* The method has a strong tendency to **degenerate to** . If is , we will only need to generate a few numbers to reach .
* It is **not random**. The next number can be easily generated from the current one. However, this issue exists in all arithmetic generators.

## Linear Congruential Generators

In **Linear Congruential Generators** (LCG), a random number , is found using the following formula:

, the **seed** value, must be provided. Here, is the **maximum** possible value that will be achievable. No can be greater than .

Thus, the formula can be re-written as:

Using this formula, we can show that

Thus, we can calculate the th random number directly instead of going iteratively.

The actual **random number** is calculated as:

Thus, can only be rational values in the set . depends on the choice of , , and .

LCG generators that use the formula above are called **mixed LCG**.

## Full Cycle Generators

In an LCG, consider that , , and .

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

Notice that, on the th attempt, the value of is , the seed value. Thus, we got a **cycle** of length . This happened because we chose to be . Every number between and is present in the cycle.

A random number generator that works like this, that has a fixed cycle length depending on its parameters, is called a **full cycle generator**.

However, not all LCG generators are full cycle generators. Whether or not we get a full cycle generator depends on our choices of **parameters**. We need to choose the parameters carefully so as to ensure we do get a full cycle generator.

A full cycle generator uses every integer between and . This ensures that the numbers we get are **uniformly distributed**, even though a part of the entire cycle might not be. Another issue is that we might sometimes get a **huge gap** between one number and the next

An LCG generator is a full cycle generator if and only if:

* is relatively prime to , meaning the only **positive integer** that **divides** both and is (the GCD).
* If is a prime number that divides , then must also divide .
* The equation must hold for all integer values of . Thus, if divides , then must also divide .

### Choosing Parameters

A **large** value for ensures we get a **long cycle** with **high density**. A reasonable choice is , where is the word-length of the computer. For a 32-bit word, , since the leftmost bit is the sign bit.

Choosing avoids **explicit division**. Say we have , , and . Using this,

Since the mod is , we do not actually need to divide. We can just take the right-most 2 bits. This example is for decimal numbers, but a similar thing happens in binary if we take . Since we do not actually divide and just take the rightmost digits, the process is fast and **efficient**.

If , is **even**. Thus, can be any **odd** number and it will be relatively prime to .

should be **divisible by** .

## Multiplicative Generators

In an LCG, consider that . Such a generator is called a **Multiplicative Generator**.

Since , the equation becomes simpler:

The challenges however, are:

* A multiplicative generator cannot be a **full cycle generator**. However, we can have a cycle of size .
* If , the **cycle length** is at most , given that is odd and or .
* From the above points, we do not know where the integers generated will fall. We might get **large gaps** and **non-uniformity**.

For , and :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

## Prime Modulus Multiplicative LCG

Instead of using in a **multiplicative generator**, if is a **prime number** less than then we have a few benefits.

For example, say . This will give us a **cycle** of length , given that is a **primitive element module** of , meaning the smaller integer, , for which is divisible by is . This is called a **prime modulus multiplicative LCG** (PMMLCG).

## More General Congruence

Instead of defining just **one seed** value, we can define **multiple**. Thus,

Here, is a deterministic function of previous values of . For LCG, .

The values will still be between and .

### Quadratic Congruential Generator

A **Quadratic Congruential Generator** is when .

### Multiple Recursive Generator

A **Multiple Recursive Generator** is when . The **cycle** resulting from this will be as large as .