Performance of Computer System

Radom Number Generation

Dr. Lina Xu
lina.xu@ucd.ie

School of Computer Science,
University College Dublin

September 25, 2018

Random Number

Why?

- Simulation must generate random values for variables in a specified random distribution, such as normal, exponential distribution

How? Two steps:

- **random number generation:** generate a sequence of uniform Full-period random numbers in [0,1]
- **random variate generation:** transform a uniform random sequence to produce a sequence with the desired distribution
  - A random variate generation algorithm is any program that halts and exits with a real number x. This x is called arandom variate.

Using Random Number: Case Study

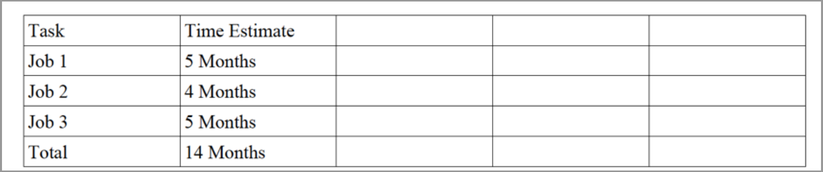
Monte Carlo simulation:

- Monte Carlo simulation, or probability simulation, is a technique used to understand the impact of risk and uncertainty in financial, project management, cost, and other forecasting models.
- When you develop a forecasting model (any model that plans ahead for the future), you make certain assumptions.
- These might be assumptions about the investment return on a portfolio, the cost of a construction project, or how long it will take to complete a certain task.
- Because these are projections into the future, the best you can do is estimate the expected value.
- For example:
    - We are estimating the total time it will take to complete a particular project.

Using Random Number: Case Study

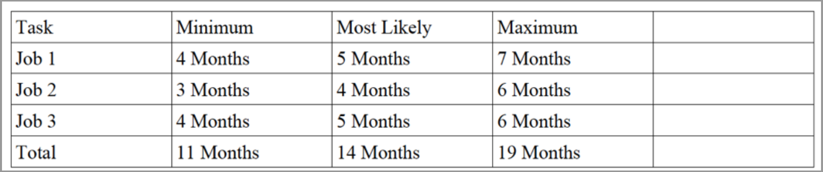
We are estimating the total time it will take to complete a particular project.

Basic Forecasting Model



Task	Time Estimate			
Job 1	5 Months			
Job 2	4 Months			
Job 3	5 Months			
Total	14 Months			

Forecasting Model Using Range Estimates



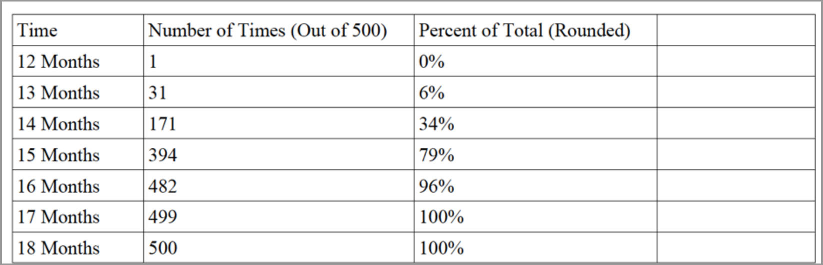
Task	Minimum	Most Likely	Maximum	
Job 1	4 Months	5 Months	7 Months	
Job 2	3 Months	4 Months	6 Months	
Job 3	4 Months	5 Months	6 Months	
Total	11 Months	14 Months	19 Months	

Using Random Number: Case Study

We are estimating the total time it will take to complete a particular project.

Monte Carlo simulation:

- We will **randomly generate values** for each of the tasks, then calculate the total time to completion.
- The simulation will be run 500 times.

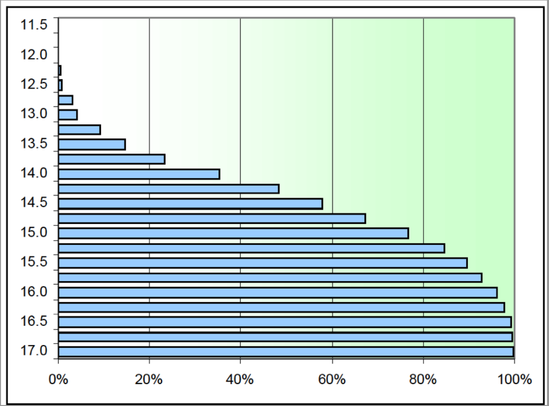


Time	Number of Times (Out of 500)	Percent of Total (Rounded)	
12 Months	1	0%	
13 Months	31	6%	
14 Months	171	34%	
15 Months	394	79%	
16 Months	482	96%	
17 Months	499	100%	
18 Months	500	100%	

Using Random Number: Case Study

We are estimating the total time it will take to complete a particular project.

Monte Carlo simulation: Probability of Completion Within Specified Time (Months)



How can I generate different random Numbers from Code in C Prog?

I want to get different random values in each output file. But it always gives same output files. How can I get different values?

- I suggest that instead of using the system time to seed multiple sequences of pseudorandom numbers you predefine the number of seeds that you need, store them in an array, and use them in sequence. The advantage of doing so is that you can reproduce the whole run. For example, it could be that one sequence out of thousand[s] causes the program to fail. If you use the system time to seed the sequences, you will not be able to reproduce and investigate that particular sequence.

How Random Number Generators Work

Most commonly use recurrence relation

- $x_n = f(x_{n-1}, x_{n-2} \dots)$
- recurrence is a function of last 1 (or a few numbers), e.g. $x_n = (5x_{n-1} + 1) \; mod \; 16$

Example: For x0= 5, first 32 numbers are

- 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5, 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4, 5
- x_n are integers in $[0, 16]$!
- Dividing by 16, get random numbers in interval $[0, 1]$

Properties of pseudo-random number sequences

Function f is deterministic

- Given the seed, we can tell with a probability of 1 what the numbers in the sequence would be.

Reproducibility (often desirable)

- Be able to repeat a simulation experiment

They pass statistical tests for randomness

- Wald-Wolfowitz test
- Contains no recognisable patternsor regularities
  - Dice roll

Wald-Wolfowitz test

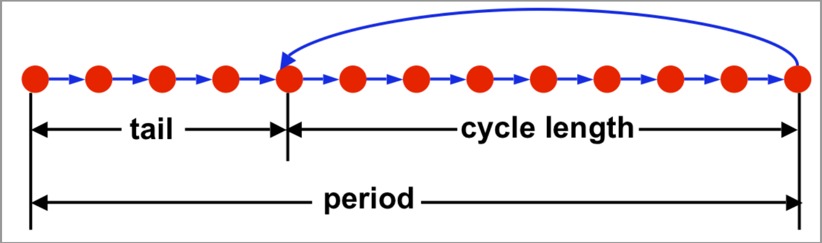
- Named after two mathematicians
- Also known as the Runs test for randomness
- $Z = (R - E(R))/sqrt(V(R))$
- $E(R) = (2nm/(n+m)) + 1$
- $V(R) = (2nm(2nm - n - m))/((n+m)^2(n+m-1))$
- Where R is number of runs, n is the number of positive values and m is the number of negative values.

Pseudo-random and fully random numbers

- Pseudo-random generator generated sequence is not truly random, because it is completely determined by an initial value
- Careful mathematical analysis is required to have any confidence that a Pseudo-random generates numbers that are sufficiently close to random to suit the intended use
- The digits in the decimal expansions of irrational number s such as π, e (the natural-logarithm base), or the square roots of numbers that are not perfect squares are believed by some mathematicians to be truly random.

Random Number Sequences

Some generators do not repeat the initial part of a sequence



Desired Properties of a Good Generator

Efficiently computable

- Since simulations typically require several thousand random numbers in each run, the processor time required to generate these numbers should be small.

Period should be large

- A small period may cause the random-number sequence to recycle, resulting in a repeated event sequence. This may limit the useful length of simulation runs.

Successive values should be

- The correlation between successive numbers should be small. Correlation, if significant, indicates dependence
- Independent
- Uniformly distributed

Linear-Congruential Generators

1951: D.H. Lehmer found that residues of successive powers of a number have good randomness

$$x_n = a^n \bmod m; \quad \text{after computing } x_{n-1}, \; x_n = \underbrace{a}_{\textbf{multiplier}} x_{n-1} \bmod \underbrace{m}_{\textbf{modulus}}$$

Many of the currently used random-number generators are a generalisation of the Lehmer's proposal

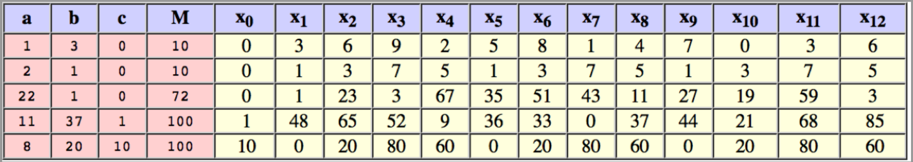
Mixed Linear-Congruential Generators

- Lehmer's generator: multiplicative LCG
- Modern generalization: mixed LCG
  - $x_n = (ax_{n-1} + b)\ mod\ m$
  - $a, b, m > 0$
- Result: x_n are integers in $[0, m-1]$
- Popular because
  - Analysed easily
  - Certain guarantees can be made about their properties

Question: The choice of a, b, and m affects the sequence?

Linear-Congruential Generators: Examples

$$x_n = (ax_{n-1} + b)\ mod\ \ m$$



a	b	c	M	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}	x_{12}
1	3	0	10	0	3	6	9	2	5	8	1	4	7	0	3	6
2	1	0	10	0	1	3	7	5	1	3	7	5	1	3	7	5
22	1	0	72	0	1	23	3	67	35	51	43	11	27	19	59	3
11	37	1	100	1	48	65	52	9	36	33	0	37	44	21	68	85
8	20	10	100	10	0	20	80	60	0	20	80	60	0	20	80	60

Properties of LCGs

- Choice of a, b, m affects
  - Period
  - Autocorrelation
- What is autocorrelation?
  - To detect non-randomness in data.
  - To identify an appropriate time series model if the data are not random.
  - Autocorrelation is a correlation coefficient. However, instead of correlation between two different variables, the correlation is between two values of the same variable at times x_i and $x_i + k$.
  - Range of values from +1 to -1. A value of 0 indicates that there is no association
  - Given measurements, $y_1, y_2, ..., y_n$ at time $x_1, x_2, ..., x_n$, the lag k autocorrelation function is defined as

$$r_k = \frac{\sum_{i=1}^{N-k}(Y_i - \bar{Y})(Y_{i+k} - \bar{Y})}{\sum_{i=1}^{N}(Y_i - \bar{Y})^2}$$

Properties of LCGs

- Observations about LCGs
  - period can never be more than m –> modulus m should be large
  - $m = 2^k$ yields efficient implementation by truncation
- If b is non-zero, obtain full period of m if and only if
  - m and b are relatively prime – no common factors
  - every prime that is a factor of m is also a factor of $a - 1$
  - if m is a multiple of 4, $a - 1$ must be too
- All of these conditions are met if
  - $m = 2k$, for some integer k
  - $a = 4c + 1$, for some integer c
  - b is an odd integer

LCGs – Good Practice

- In practice, we often want random integers in a small range, say between 0 and 9. Typically we do so by using a large M in a linear congruential random number generator, and then use arithmetic to bring them down into a small range.
- For example: $a = 11, b = 37, c = 1, m = 100$
  - Range 1-00: 1 48 65 52 9 36 33 0 37 44 21 68 85 72 29 56 53 20 57 64 41 88 5 92 49 76 73 40 77 84 61 8 25 12 69 96 93 60 97 4 81 28 45 32 89 16 13 80 17 24 1 48 65 52 9 36 33 0 37 44 21 68 85 72 29 56 53 20 57 64 41 88 5 92 49 76 73 40 77 84 61 8 25 12 69 96 93 60 97 4 81 28 45 32 89 16 13 80 17 24
  - Most left digital: 0 4 6 5 0 3 3 0 3 4 2 6 8 7 2 5 5 2 5 6 4 8 0 9 4 7 7 4 7 8 6 0 2 1 6 9 9 6 9 0 8 2 4 3 8 1 1 8 1 2
  - Most right digital: 1 0 2 8 6 0 2 8 6 0 2 8 6 0 2 8 6 0 2 8 6 0 2 8 6 0 2 8 6 0 2 8 6 0 2 8 6 0 2 8 6 0 2 8 6 0 2 8 6 0

Properties of LCGs – Continue...

- Full-period generator = one with period m
  - Not all are equally good
  - Lower Autocorrelation between adjacent elements is better

Example: Two Candidate LCGs

- Examples

$$x_n = ((2^{34} + 1)x_{n-1} + 1) \bmod 2^{35}$$

$$x_n = ((2^{18} + 1)x_{n-1} + 1) \bmod 2^{35}$$

Both must be full period generators

- $m = 2k$, for some integer k
- $a = 4c + 1$, for some integer c
- b is an odd integer

Autocorrelation for the first one is 0.25, for the second one is 2^{-18}

Multiplicative LCGs

- More efficient than mixed LCGs: no addition
- Two classes: $m = 2^k$ and $m\ ! = 2^k$

$$x_n = a^n\ mod\ m$$

Multiplicative LCG with $m = 2^k$

- Most efficient LCG: mod = truncation
- Not full-period: maximum possible period for $m = 2^k$ is 2^{k-2}
  - only possible if multiplier $a = 8i +/- 3$ and x_0 (initial seed) is odd
  - consider

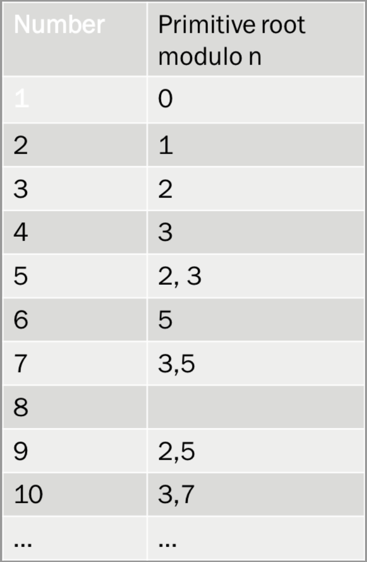
    $x_n = 5^n \ mod \ 2^5$

    $x_n = 7^n \ mod \ 2^5$
- If 2^{k-2} period suffices, may use multiplicative LCG for efficiency !

Multiplicative LCG with $m \, ! = 2^k$

- Avoid small period of LCG when $m \, ! = 2^k$: use prime modulus
- Full period generator with proper choice of a, let a to be a **primitive root** of the modulus m
  - when a is primitive root of m, a^n mod m !=1 for n = 1, 2, ..., m-2
- Consider
  - $x_n = 3^n \ mod$ 31:
    1, 3, 9, 27, 19, 26, 16, 17, 20, 29, 25, 13, 8, 24, 10, 30, 28, 22, 4, 12, 5, 15, 14, 11, 2, 6, 18, 23, 7, 21, 1...
  - $x_n = 5^n \ mod$ 31:
    1, 5, 25, 1...
- Note: 53 mod 31=125 mod 31=1

Primitive Root Table



Number	Primitive root modulo n
1	0
2	1
3	2
4	3
5	2, 3
6	5
7	3,5
8	
9	2,5
10	3,7
…	…

For example, 31: the primitive roots are 3, 11, 12, 13, 17, 21, 22, 24

Random-Numbers in Java

```
// 25214903917
private final static long multiplier = 0x5DEECE66DL;
// 11
private final static long addend = 0xBL;
// 248-1 = 281474976710655
private final static long mask = (1L << 48) - 1;
protected int next(int bits) {
        long oldseed, nextseed; ...
        oldseed = seed.get();
        nextseed = (oldseed * multiplier + addend) & mask;
        ...
        return (int)(nextseed >>> (48 - bits));
}
```

Other Generators

- Tausworthe generators
- Extended Fibonacci generators
- Combined generators