Random numbers generation

Dr. Ahmad S. Tarawneh

Outline

- Random numbers properties
- Generation of Pseudo-Random numbers
- Methods of Pseudo-Random numbers generation
- Testing random numbers

Overview

- Randomness and random numbers are essential for fields like simulation, games, cryptography, some machine learning methods, etc.
- In simulation stochastic systems, we need to frequently generate random numbers. For example, random number for interarrival time, or service time

Cont.

- A random number generator (RNG) is any mechanism that produces **independent** random numbers
- Independent means that the probability of generating a random number will not affect the generation, or occurrence, of another random number.
- For example, in rolling a dice, the number might be 3 with a probability of 1/6, rolling a dice a second time does not change the fact that number 3 has the same probability to occur.

Properties of RN

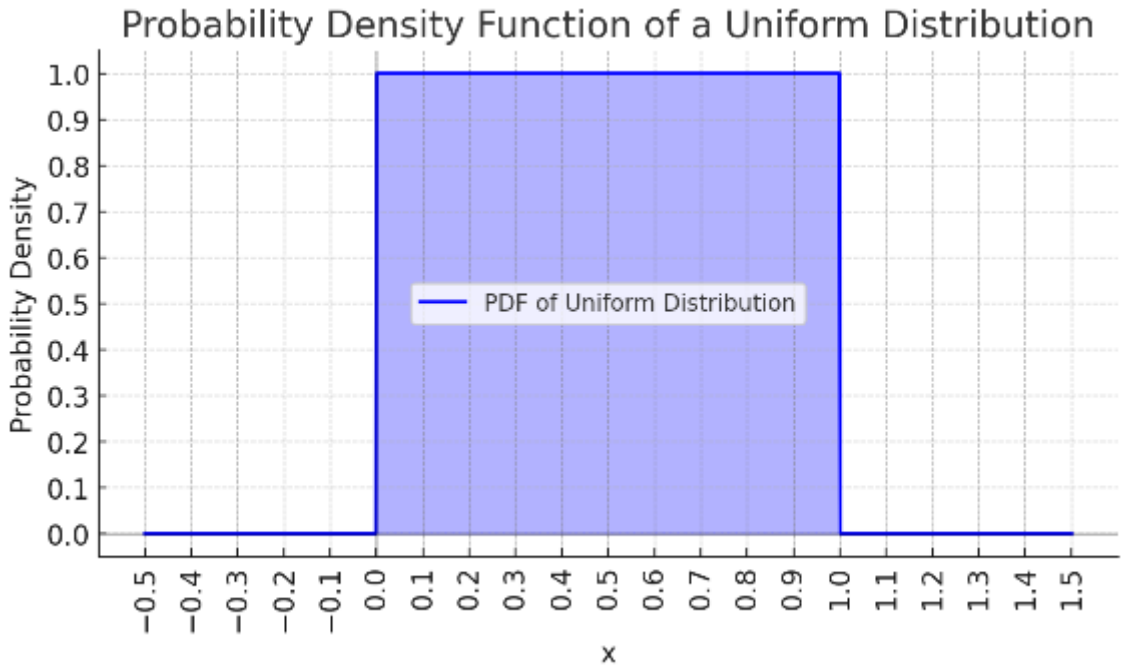
- A sequence of random numbers, $R_1, R_2, \ldots, R_n$ must have two main properties

1. Uniformity: The number should be drawn from the uniform distribution; each number has the same probability as other numbers.
2. Independence: the generation of one number is not affected by the generation of another

Cont.

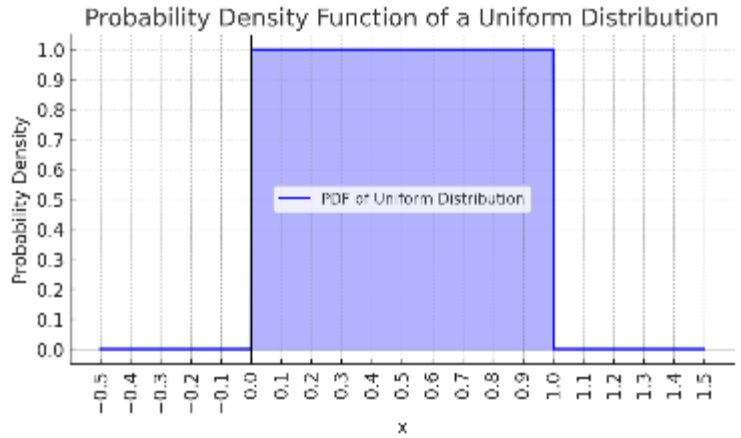
- Therefore, a random number must be independently generated from a uniform distribution, with a probability density function (PDF)

$$f(x) = \begin{cases} \frac{1}{b-a} & for\ a \le x \le b \\ 0 & otherwise \end{cases}$$



Cont.

- When a is 0 and b is 1, we call this a standard uniform distribution



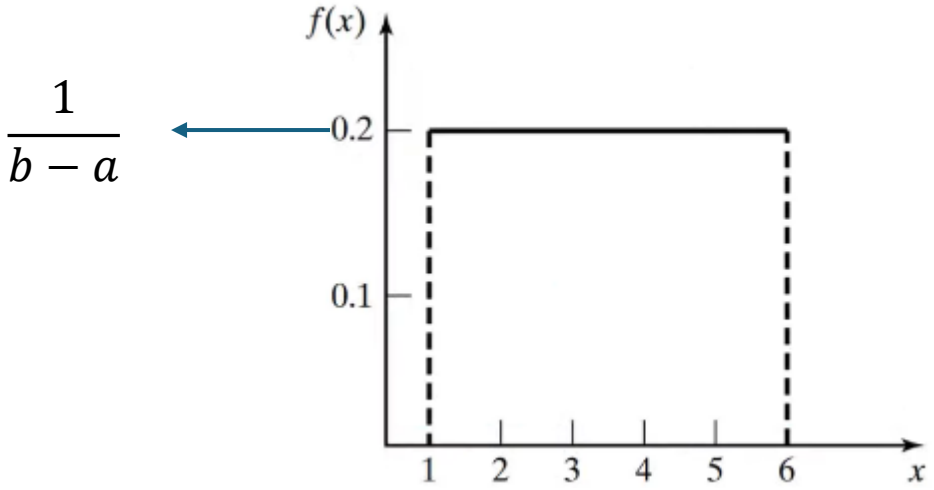
- Each number in this range has the same probability to occur.
- The expected value, mean, and variance are given as follows

$$E(X) = \mu = \frac{a+b}{2}$$

$$V(X) = Varience = \frac{(b-a)^2}{12}$$

Example

- what is the probability of occurrence of random numbers in the range 1-6
  - a = 1
  - b = 6



- The expected value is $\frac{1+6}{2} = \frac{7}{2} = 3.5$
- The variance is $\frac{(b-a)^2}{12} = \frac{(6-1)^2}{12} = \frac{25}{12} = 2.08$

Typically, we generate continuous numbers between 0 and 1 and then factorize them, transform them to any range.

Independence

- In true random numbers, the next generated number is **unpredictable** => usually using hardware
- This process of creating truly random numbers is non-deterministic
- Computers are deterministic, they perform operation depending on known rules
- Random numbers generated by functions are deterministic and predictable, therefore we call them false random numbers, or pseudo-random numbers (PRNs)
  - However, they appear random as they follow the random number rules
  - Uniformity and independence

Generation of PRNs

- There are many methods used to generate PRNs. However, there are characteristics of each method
- The PRNG methods are evaluated based on several criteria:
  - Space & time complexity: should be fast and does not require large memory to work
  - Portability: should not produce different numbers on different platforms
  - Replicability: should be able to generate the same sequence of RN if needed, given seed (starting point)
  - Cycle: how many RNs can the method generate before the numbers start to repeat themselves

PRNG methods

- There are many methods for PRNs generation, follows are some of them:
  1. Middle-Squares (*midsquare*) 1949
  2. Linear-Congruential Generation 1958
  3. Mersenne Twister (MT) 1998
  4. Philox 2011
  5. Squares RNG 2020
- A list of over than thirty methods is available on Wikipedia
  - https://en.wikipedia.org/wiki/List_of_random_number_generators

Midsquare

- One of the simplest methods for PRN generation
- To generate a sequence of n-digits PRNs, a seed of n-digits is used and squared to produce a 2n-digits number.
  - n should be an even number
- If the number is less than 2n-digits, the number is pre-padded with zeros
- The middle part of the number is taken as the result, first PRN, and this can be used to generate the next seed
- This process continuous to generate more numbers

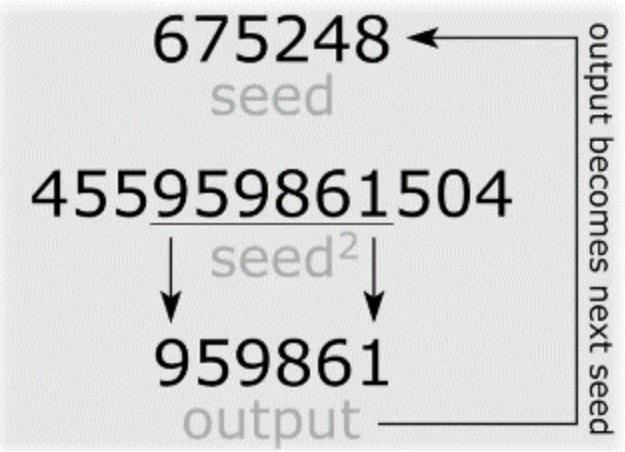
Pseudocode

```
Procedure MiddleSquareMethod(seed, n)
    Begin
        currentSeed := seed

        Repeat
            // Step 1: Square the current seed
            squared := currentSeed * currentSeed

            // Step 2: Convert the squared number to a string and pre-pad with zeros if necessary
            squaredString := ConvertToStringAndPad(squared, 2*n)

            // Step 3: Extract the middle n digits
            middleDigits := ExtractMiddleDigits(squaredString, n)

            // Step 4: Convert the middle digits back to a number
            nextSeed := ConvertToNumber(middleDigits)

            // Output or use the generated pseudo-random number
            Print(nextSeed)

            // Prepare for next iteration
            currentSeed := nextSeed
        Until stopping condition is met
    End
```

Example

- Assume we start with the seed 675248

- First, we square this number => 675248^2
  - The result of squaring the number cannot have more digits than $2n$,
  - n is 6 in our case, the result cannot have digits more than 12
    - **it can have less**
- We take the n-digit number in the middle
  - **959861**
- Now this number is our new seed



Example 2

- Consider the following: If a 3-digit number is squared it can yield a 6-digit number (eg: $540^2 = 291600$).
- If there were to be a middle three digit that would leave $6 - 3 = 3$ digits to be distributed to the left and right of the middle.
- It is impossible to distribute these digits **equally on both sides of the middle number** and therefore there are no 'middle digits'.
- It s acceptable to pad the seeds with zeros to the left to create an even-valued n-digit (eg: $540 -> 0540$).
  - Pad the odd number (seed) with 0 to make it even
  - or $291600 -> 0291600$
    - 916 next seed

Notes about Midsquare

- For an n-digit numbers the maximum cycle length is 8^n
- If it happens that the middle n-digits are zeros, the rest of the random numbers are zeros
- if the first half of the middle n-digits are zeros, the numbers will be decreasing to zeros
- Midsquare can stuck with numbers like 0100, 2500, 7600
  - e.g., 0100 -> 00010000 -> 000100000 -> 0100 -> .....
- Some numbers give short cycle
  - e.g., **0540** -> 2916 -> 5030 -> 3009 -> **0540**
  - Selecting the seed is very important in this method

Example

Example with seed = 7182

PRNs

i	Z_i	U_i	Z_i^2
0	7182	—	51,581,124
1	5811	0.5811	33,767,721
2	7677	0.7677	58,936,329
3	9363	0.9363	87,665,769
4	6657	0.6657	44,315,649
5	3156	0.3156	09,960,336
.	.	.	.
.	.	.	.
.	.	.	.

- This is a 4-digits
- U_i can be calculated as follows

$$U_i = \frac{Z_i}{the\ \max number\ we\ can\ get\ with\ \boldsymbol{n\ digits} + 1}$$

- The maximum number we can get with 4 digits is 9999
  - then we divide by 10000
  - or $10^{n-digits}$
  - $\frac{5811}{10000} = 0.5811$

Linear Congruential Generators

Linear Congruential Generators

- This method works recursively to generate a sequence of seeds $Z_1, Z_2, Z_3 \ldots Z_n$ based on the following formula:

$$Z_i = (aZ_{i-1} + c)\ mod\ m$$

- where
  - a is called the multiplier => nonnegative integer less than m
  - c is called the increment => nonnegative integer less than m
  - m is the modulus => should be a prime number or power of a prime number
    - e.g., 7, 7^4, 5^2 … etc.
  - we start with $Z_{i-1} = Z_0 \Rightarrow$ nonnegative integer
- good selection of these parameters ensures long cycle length

Example

The LCG $Z_i = (5Z_{i-1} + 3)(\text{mod } 16)$ with $Z_0 = 7$

Example

i	Z_i	U_i	i	Z_i	U_i	i	Z_i	U_i	i	Z_i	U_i
0	7	—	5	10	0.625	10	9	0.563	15	4	0.250
1	6	0.375	6	5	0.313	11	0	0.000	16	7	0.438
2	1	0.063	7	12	0.750	12	3	0.188	17	6	0.375
3	8	0.500	8	15	0.938	13	2	0.125	18	1	0.063
4	11	0.688	9	14	0.875	14	13	0.813	19	8	0.500

m is so small; it only illustrates the arithmetic of LCGs

We calculate U_i by dividing the output by m

$$U_i = \frac{Z_i}{m}$$

The maximum number of PRNs is m-1, that is the full period

Code

- Python code for the previous example

```
# Parameters
seed = 7  # Initial seed
a = 5  # Multiplier
c = 3  # Increment
m = 16  # Modulus
n = 19  # Number of random numbers to generate

def lcg(seed, a, c, m, n):
    random_numbers = []
    x = seed
    for _ in range(n):
        x = (a * x + c) % m
        random_numbers.append(x)
    return random_numbers

# Generate pseudorandom numbers
r_numbers = lcg(seed, a, c, m, n)

[x/m for x in r_numbers]
```

Longer cycle length

```
# Parameters
seed = 42  # Initial seed
a = 1664525  # Multiplier
c = 1013904223  # Increment
m = 2**32  # Modulus
n = 80  # Number of random numbers to generate

def lcg(seed, a, c, m, n):
    random_numbers = []
    x = seed
    for _ in range(n):
        x = (a * x + c) % m
        random_numbers.append(x)
    return random_numbers

# Generate pseudorandom numbers
r_numbers = lcg(seed, a, c, m, n)

[x/m for x in r_numbers]
```

Notes

- The LCG can give the full period iff the following conditions are met:

1. if m is relatively prime to the increment c
   - ❖ The greatest common devisor gcd(m, c) is 1

Example with Numbers 12 and 8
Let's find the GCD of 12 and 8.
1.List the divisors of each number:
1. Divisors of 8: 1, 2, 4, 8
2. Divisors of 12: 1, 2, 3, 4, 6, 12

2.Identify the common divisors:
1. Common divisors of 8 and 12 are: 1, 2, 4

3.Find the greatest common divisor:
1. The largest number in the list of common divisors is 4

12 and 8 are not relatively prime

Example with Numbers 28 and 9
Let's find the GCD of 28 and 9.
1.List the divisors of each number:
1. Divisors of 9: 1, 3, 9
2. Divisors of 28: 1, 2, 4, 7, 14, 28

2.Identify the common divisors:
1. The only common divisor of 9 and 28 is 1.

3.Determine the greatest common divisor:
1. Since the only common divisor is 1, the GCD of 9 and 28 is 1.

28 and 9 are relatively prime

Cont.

2. If q is a prime number that divides m, then q should divide $a - 1$

3. If 4 divides m, then 4 should divides $a - 1$

The parameter ***c***

- If c is greater than 0, we call it mixed LCGs
- If c is equal to 0, we call it multiplicative LCGs

❑Mixed: For a large period and high density of the Ui's on [0, 1], we want m to be large. A choice of m that is good in all these respects is $m = 2^b$, where b is the number of bits (binary digits) in a word on the computer being used that are available for actual data storage.

❑For example, most computers and compilers have 32-bit words, the leftmost bit being a sign bit, so b = 31 and $m = 2^{31}$ > 2.1 billion.

Multiplicative

- Multiplicative LCGs cannot have full period since the first condition cannot be satisfied. As we shall see, however, it is possible to obtain period (m —1)if m and a are chosen carefully. (Note: m is a prime)

- As with mixed generators, it's still computationally efficient to choose m = 2^b . However, it can be shown that in this case the period is at most 2^{b-2} , that is, m/4.

Cont.

- Using the multiplicative congruential method, find the period of the generator for $a = 13, m = 2^6 = 64$ and $X_i = 1,2,3\ and\ 4$.

$$\frac{m}{4} = \frac{64}{4} = 16$$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Xi	1	13	41	21	17	29	57	37	33	45	9	53	49	61	25	5	1
Xi	2	26	18	42	34	58	50	10	2								
Xi	3	39	59	63	51	23	43	47	35	7	27	31	19	55	11	15	3
Xi	4	52	36	20	4												

Seed selection

- In general, the seed value used to initialize an RNG should not affect the results of the simulation.
- However, a wrong combination of a seed and a random generator may lead to erroneous conclusions.
- If the RNG has a full period and only one random variable is required, any seed value is as good as any other.
- However, care is required in choosing seeds for simulation requiring random numbers for more than one variable. Such types of simulations are often called multistream simulations.

Cont.

- Most simulations are multistream simulations.
- For example, simulation of a single queue such as a bank with a single teller requires generating random arrival and random service times.
- This simulation would require two streams of random numbers: one for interarrival times and other for service times.

Recommendations for seed selection

- Do not use zero.
  - This can make certain generators produce a sequence of zeros.
- Do not use randomly selected seeds.
- Do not use the same seed and the same random number generator to obtain samples of different input parameters (for example arrival times and service times) because the resulting samples may be strongly correlated.
- Avoid using even values as seeds
  - This results to a decrease in the period of certain random number generators.
- When replicating a simulation run, use different seeds.