UM-SJTU JOINT INSTITUTE

Probabilistic Methods in Engineering

(VE401)

Term Project 1

Randomness and Pseudorandom Numbers

Name: Feitong Tang	ID: 518370910017	Group 1
Name: Weikai Zhou	ID: 518021911039	Group 1

Date: 3 April 2020

1 Abstract

This report introduces some basic knowledge of pseudorandom numbers and pseudorandom number generators. It describes basic ways of generating pseudorandom numbers, including linear congruential method, nonlinear congruential method and additive congruential method. Moreover, it uses linear congruential method as an example to design a simple algorithm to generating pseudorandom numbers.

Contents

2 Introduction

2.1 Background

Pseudorandom numbers are number sequences generated by certain computer algorithm that behave like random numbers. A pseudorandom number generator (PRNG) is the algorithm designed for producing numbers can be regarded as random numbers. They are of great significant in our daily lives. They can be used in various aspects including simulations, electronic games and cryptography [1]. However, due to the limitations of algorithms, the numbers are not truly random, which may give people chances to predict the next number from the previous. Especially, the cryptographic applications need almost perfect pseudorandom numbers, otherwise, people can decode confidential information easily. Therefore, elaborate PRNGs are in demand nowadays.

2.2 Objectives

- Try to understand some approaches that can be used to obtain pseudorandom numbers;
- Choose one of the approaches to create a feasible PRNG algorithm;
- Try to understand some approaches to testing the randomness of pseudorandom numbers;
- Test the randomness of the pseudorandom numbers generated by the PRNG algorithm;
- Try to find some flaws of the PRNG.

3 Ways of obtaining pseudorandom numbers

There are different ways of obtaining pseudorandom numbers, and the project will basically introduce linear congruential method, nonlinear congruential method and additive congruential method.

3.1 Linear congruential method

This is one of the most popular methods to generate pseudorandom numbers proposed by Lehmer in 1951. The general form of it can be written as

$$x_{n+1} = (ax_n + c) \bmod m. \tag{1}$$

Here $\{x_n\}$ is the sequence of pseudorandom numbers. a is an integer greater than 1, known as the multiplier. c is an integer known as the increment, and m is the integer known as the modulus. And "$(ax_n + c) \bmod m$" means the remainder of dividing $(ax_n + c)$ by m. The initial integer x_0 is known as the seed, and it is usually in the interval $[\,0, m\,)$. Therefore, all x_n are in the interval $[\,0, m\,)$, and by dividing m, all x_n can be projected into the interval $[\,0, 1\,)$ [2].

Especially, if $c = 0$, Eq. (1) can be written as

$$x_{n+1} = ax_n \bmod m, \tag{2}$$

which is known as the multiplicative congruential generator. Besides, if $c \neq 0$, people tend to call Eq. (1) mixed congruential generator [3].

3.2 Nonlinear congruential method

In 1981, Knuth proposed the nonlinear congruential generator. The general form of it can be written as

$$x_{n+1} = (dx_n^2 + ax_n + c) \bmod m \quad (d \neq 0). \tag{3}$$

Here $\{x_n\}$ is the sequence of pseudorandom numbers, and m is the integer known as the modulus. In 1988, Eichenauer et al. came up with a more general form for nonlinear congruential generators

$$x_{n+1} = f(x_n) \bmod m, \tag{4}$$

where the function $f(x_n)$ can be any functions other than linear function [2]. Also, by dividing m, all x_n can be mapped into the interval $[\,0, 1\,)$.

3.3 Additive congruential method

The kth order additive congruential random number generator needs to define an integer m as the modulus, an integer x_0^0 as the seed with the property that $x_0^0 \in (0, m)$ and an arbitrary set of k integers $\{x_0^i\}$ where $i = 1, 2, 3, ..., k$ and $x_0^i \in [0, m)$. They further satisfy the following two equations:

$$x_n^0 = x_{n-1}^0 \quad (n \geq 1), \tag{5}$$

$$x_n^i = \left(x_n^{i-1} + x_{n-1}^i\right) \bmod m \quad (n \geq 1,\ i = 1, 2, 3, ..., k), \tag{6}$$

where "$\left(x_n^{i-1} + x_{n-1}^i\right) \bmod m$" means the remainder of dividing $x_n^{i-1} + x_{n-1}^i$ by m [4].

Moreover, the explicit form of x_n^i is given by Eq. (7) after Wikramaratna's analysis [5]:

$$x_n^i = \left(\sum_{p=0}^{i} x_0^p z_n^{i-p}\right) \bmod m, \tag{7}$$

where

$$z_n^{i-p} = \frac{(n+i-p-1)!}{(n-1)!(i-p)!}. \tag{8}$$

And we can dividing x_n^i by m to map them into $[0, 1)$.

4 A feasible PRNG algorithm

The project is going to choose the linear congruential method as an example to create a feasible PRNG algorithm in C so that it can produce one hundred pseudorandom numbers in the interval $[0, 100)$.

4.1 Designing process

As described in 3.1, we can let $a = 2$, $c = 3$. m can be set as 100 since the interval of the pseudorandom numbers is $[0, 100)$. Then an array that can store one hundred numbers is needed. For the starting point or the seed of the PRNG, the function "time(NULL)" provided by C can be used. It uses the current time to initialize "srand()". Afterwards, the first pseudorandom number can be achieved with "x [0] = rand() % 100". The "%" here is like the function of "mod", which can get the remainder. Furthermore, a "for-loop" should be used to repeat "x [i + 1] = (a * x [i] + c) % 100" 99 times to get one hundred pseudorandom numbers. Finally, we can use "printf" to output all the numbers we get. The detailed code is attached in Appendix.

4.2 Demonstration and self-examination

With the code, we can run a test case. The pseudorandom numbers generated are listed below (Table 1).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	80	63																		
2	29	61	25	53	9	21	45	93	89	81	65	33	69	41	85	73	49	1	5	13
3	29	61	25	53	9	21	45	93	89	81	65	33	69	41	85	73	49	1	5	13
4	29	61	25	53	9	21	45	93	89	81	65	33	69	41	85	73	49	1	5	13
5	29	61	25	53	9	21	45	93	89	81	65	33	69	41	85	73	49	1	5	13
6	29	61	25	53	9	21	45	93	89	81	65	33	69	41	85	73	49	1		

Table 1: Pseudorandom numbers generated by the first trial.

From this table, we can easily find that there is a loop in the numbers generated. The reason for this could be either the problem of x [0], or the problem of parameters a, c and m. We test it again, and the new pseudorandom numbers are listed below (Table 2).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	73	49	1	5	13	29	61	25	53	9	21	45	93	89	81	65	33	69	41	85
2	73	49	1	5	13	29	61	25	53	9	21	45	93	89	81	65	33	69	41	85
3	73	49	1	5	13	29	61	25	53	9	21	45	93	89	81	65	33	69	41	85
4	73	49	1	5	13	29	61	25	53	9	21	45	93	89	81	65	33	69	41	85
5	73	49	1	5	13	29	61	25	53	9	21	45	93	89	81	65	33	69	41	85

Table 2: Pseudorandom numbers generated by the second trial.

The loop is still there, therefore, the problem may be the value of a, c and m. For example, if m is small, then the values of remainders are limited, which increases the chance of falling into a loop like shown above. Hence, it is very important to choose appropriate values of a, c and m. Some good values used by different platforms are listed below (Table 3).

Source	modulus m	multiplier a	increment c
Numerical Recipes	2^{32}	1664525	1013904223
Borland C/C++	2^{32}	22695477	1
glibc (used by GCC)	2^{31}	1103515245	12345
ANSI C: Watcom, Digital Mars, CodeWarrior, IBM VisualAge C/C++; C90, C99, C11: Suggestion in the ISO/IEC 9899, C18	2^{31}	1103515245	12345
Borland Delphi, Virtual Pascal	2^{32}	134775813	1
Turbo Pascal	2^{32}	134775813	1
Microsoft Visual/Quick C/C++	2^{32}	214013	2531011
Microsoft Visual Basic (6 and earlier)	2^{16}	1140671485	12820163
RtlUniform from Native API	2^{31}-1	2147483629	2147483587
Apple CarbonLib, C++11's minstd_rand0	2^{31}-1	16807	0
C++11's minstd_rand	2^{31}-1	48271	0
MMIX by Donald Knuth	2^{64}	6364136223846793005	1442695040888963407
Newlib, Musl	2^{64}	6364136223846793005	1
VMS's MTH$RANDOM, old versions of glibc	2^{32}	69069	1
Java's java.util.Random, POSIX [ln]rand48, glibc [ln]rand48[_r]	2^{48}	25214903917	11
random0	134456	8121	28411
POSIX [jm]rand48, glibc [mj]rand48[_r]	2^{48}	25214903917	11
POSIX [de]rand48, glibc [de]rand48[_r]	2^{48}	25214903917	11
cc65	2^{23}	65793	4282663
cc65	2^{32}	16843009	826366247
Formerly common: RANDU	2^{31}	65539	0

Table 3: Good parameters used by different platforms [3].

4.3 Modification

Based on Table 3, we can re-design the algorithm. Let $m = 134456$, $a = 8121$ and $c = 28411$. However, this will enlarge the interval of pseudorandom numbers to $[0, 134456)$. In order to solve this problem, we can first map them into $[0, 1)$ by dividing them with 134456, and then re-map them into $[0, 100)$ by timing them with 100. But the numbers we get now are not integers. We can use function "floor()" to round them, which will return the largest integer less than the value in the brackets. The detailed code is attached in Appendix, and the pseudorandom numbers generated are listed below (Table 4).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	13	79	56	25	82	86	1	36	90	98	65	19	47	40	38	67	90	42	79	16
2	74	19	9	33	56	61	94	1	81	17	27	7	66	79	23	22	89	73	94	93
3	73	38	42	27	69	18	43	75	47	42	31	16	79	15	85	50	22	99	15	76
4	10	3	47	81	33	64	10	27	29	38	20	32	64	81	66	64	23	39	97	16
5	89	28	6	1	37	27	18	33	12	57	74	14	22	72	14	19	16	40	71	82

Table 4: Pseudorandom numbers generated by the thrid trial.

From Table 4, we may assume that there are no rules behind the numbers. However, in order to confirm this assumption, we need to find some ways that can test the randomness of the pseudorandom numbers.

5 Ways of randomness test

6 A test for randomness of the PRNG

7 Flaw in the PRNG

8 Conclusion

9 References

[1] Wikipedia. Pseudorandom number generator. `https://en.wikipedia.org/wiki/Pseudorandom_number_generator`

[2] W. L. Dunn, and J. K. Shultis, "*Pseudorandom number generators*," Exploring Monte Carlo Methods, pp. 47-68, 2012.

[3] Wikipedia. Linear congruential generator. `https://en.wikipedia.org/wiki/Linear_congruential_generator`

[4] R. S. Wikramaratna, "*The additive congruential random number generator—A special case of a multiple recursive generator*," Journal of Computational and Applied Mathematics, vol. 216, no. 2, pp. 371-387, July 2008.

[5] R.S. Wikramaratna, "*Theoretical background for the ACORN random number generator*," AEA Technology, 1992.

10 Appendix

10.1 Code for 4.1

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
        int x[100];
        int a = 2;
        int c = 3;
        srand(time(NULL));
        x[0] = rand()%100;
        printf("%d ", x[0]);
        for (int i = 0; i < 99; i++) {
                x[i + 1] = (a * x[i] + c) % 100;
                printf("%d ", x[i + 1]);
        }
        return 0;
}
```

10.2 Code for 4.3

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

int main() {
        int x[100];
        float y[100];
        int m1 = 134456;
        float m2 = 134456;
        int a = 8121;
        int c = 28411;
        srand(time(NULL));
        x[0] = rand()% m1;
        for (int i = 0; i < 99; i++) {
                x[i + 1] = (a * x[i] + c) % m1;
        }
        for (int i = 0; i < 100; i++) {
                y[i] = x[i] / m2 * 100 ;
        }
        for (int i = 0; i < 100; i++) {
                printf("%d ", int(floor(y[i])));
        }
        return 0;
}
```