ENE 3031
Computer Simulation
Week 5: Random Number Generation
(attributed by Dr. Goldsman)

Chuljin Park
Assistant Professor
Industrial Engineering
Hanyang University

Introduction

- Goal: Give an algorithm that produces a sequence of "pseudo-random" numbers (PRN) R_1, R_2, … that "appear" to be i.i.d. Unif(0,1).
- Desired PRN generators need to have:
  - Fast speed and adaptability
  - Long cycle and ability to reproduce any sequence it generates
  - Uniformity and independence

Outline

A. PRN Generators
1. Random devices
2. Table of random numbers
3. Midsquare (not useful)
4. Fibonacci (not useful)
5. Linear Congruential
6. Tausworthe
7. Hybrid

B. Choosing a Good Generators
1. Uniformity
2. Independence

1. Random Devices

Nice randomness properties.

However, Unif(0,1) sequence storage difficult, so it's tough to repeat experiment.

Examples:
a. flip a coin
b. particle count by Geiger counter
c. least significant digits of atomic clock

2. Random Number Tables

List of digits supplied in tables.
Cumbersome and slow — not very useful.
Once tabled no longer random.

3. Mid-Square Method (J. von Neumann)

Idea: Take the middle part of the square of the previous random number. John von Neumann was a brilliant and fun-loving guy, but method is **lousy**!!!

Example: Take $R_i = X_i/10000$, $\forall i$, where the X_i's are positive integers < 10000.

Set $X_1 = 6632$; then $6632^2 \rightarrow 43(9834)24$;
So $X_2 = 9834$. Now $9834^2 \rightarrow 96(7075)56$;
Set $X_3 = 7075$, etc,...

Unfortunately, positive serial correlation in R_i's.

Also, occasionally degenerates; e.g., consider $X_i = 0003$.

4. Fibonacci and Additive Congruential Generators

These methods are also no good!!

Take $X_i = (X_{i-1} + X_{i-2})$ mod m $i = 1, 2, \ldots$, where

$R_i = \frac{X_i}{m}$, m is the *modulus*, X_0, X_1 are *seeds*, and

$a = b \mod m$ iff a is the remainder of $\frac{b}{m}$, e.g., 6 = 13 mod 7.

Problem: Small numbers follow small numbers.

Also, not possible to get $X_{i-1} < X_{i+1} < X_i$ or
$X_i < X_{i+1} < X_{i-1}$ (which should occur w.p. 1/3).

5. Linear Congruential Generators (LCG)

Most widely used!

$X_i = (aX_{i-1} + c) \mod m$, X_0 is the seed.

$R_i = \frac{X_i}{m}$, $i = 1, 2, \ldots$

Choose a, c, m carefully to get good statistical quality and long *period* or *cycle length*, i.e., time until LCG starts to repeat itself.

If $c = 0$, LCG is called a *multiplicative* generator.

Example: Suppose $X_0 = 57$. Then

X_1 = (7x57+0) mod 100 = 399 mod 100 = 99
X_2 = (7x99+0) mod 100 = 693 mod 100 = 93
X_3 = (7x93+0) mod 100 = 51
....
So, $R_1 = 0.99, R_2 = 0.93, R_1 = 0.51$

Common choices for a, m with $c = 0$:
a = 16807, $m = 2^{31} - 1 \leftarrow$ APL, old IMSL,
where 31 = number of bits in a typical word.
a = 630360016, $m = 2^{31} - 1 \leftarrow$ SIMSCRIPT

Bad choice: a = 65539, $m = 2^{31}$
Powers of 2 were popular in 60's, but give observably bad behavior!

Induction shows:

$X_i = \left(a^i X_0 + \frac{c(a^i-1)}{a-1}\right)$ mod m

This is obviously **not** random!! But it gives X_i's that **appear** random (see L'Ecuyer and Bouin 1988).

Generalization:

$X_i = \sum_{j=1}^{q} a_i X_{i-j}$ mod m, a_i constant.

Extremely large periods (up to $m^q - 1$ possible if parameters are chosen properly).

But watch out! — Fibonacci is a special case.

Portable FORTRAN implementation (works fine and is fast):

Bratley, Fox, and Schrage (1987)
$a = 16807$, $m = 2^{31} - 1$
$X_i = 16807 X_{i-1}$ mod $(2^{31} - 1)$

Note: set 1< IX < $2^{31} - 1$, IX is an integer

```
FUNCTION UNIF(IX)
K1 = IX/127773
IX = 16807*(IX-K1*127773) - K1*2836
IF (IX.LT.0) IX = IX + 2147483647
UNIF = IX*4.656612875E-10
RETURN
END
```

UNIF is the real valued output in (0,1).

6. Tausworth Generator

Define a sequence of binary digits $B_1, B_2, \ldots$, by

$$B_i = \left(\sum_{j=1}^{q} c_j B_{i-j}\right) \mod 2,$$

where $c_i = 0$ or 1.

Looks like generalization of linear congruential.

Usual implementation (saves computational effort):

$B_i = (B_{i-r} + B_{i-q})$ mod 2, $0 < r < q$

Obtain

$B_i = 0$, if $B_{i-r} = B_{i-q}$ or
$B_i = 1$, if $B_{i-r} \neq B_{i-q}$

To initialize the B_i sequence specify $B_1, B_2, \ldots, B_q$.

Example (Law and Kelton, 2000):

$r = 3, q = 5; B_1 = \cdots = B_5 = 1$

$B_i = (B_{i-3} + B_{i-5})$ mod 2 $= B_{i-3}$ or B_{i-5}, $\quad i > 4$
$B_6 = (B_3$ or $B_1) = 0$, etc.

Turns out period of 0-1 bits is $31 = 2^q - 1$.

How do we go from B_i's to Unif(0,1)'s?

Easy way:
(l-bit binary integers)/2^l

Example:
Set $l = 4$ in previous equation:
$\frac{15}{16}, \frac{8}{16}, \cdots \to 1111, 1000, \ldots$

Lots of potential for Tausworth generators.
Nice properties — long periods, fast calculation.
Theoretical properties still being investigated.

B. Choosing a Good Generator: Theory

Look at one-step serial correlation and cycle length.

Serial Correlation of LCG's:

$$\text{Corr}(R_1, R_2) \leq \frac{1}{a}\Big(1 - \frac{6c}{m} + 6(\frac{c}{m})^2\Big) + \frac{a+6}{m}$$

(This is a loose upper bound from Greenberger, 1961.)

Cycle Length of LCG's

Goal: Try to get a full cycle generator, i.e., one that generates every integer in $[1, m-1]$ before repeating.

7. Combinations of Generators:

Suggestions on how to use $X_1, X_2, \ldots$ and $Y_1, Y_2, \ldots$ to construct $Z_1, Z_2, \ldots$

a. Set $Z_i = (X_i + Y_i)$ mod m
b. Schuffling
c. Set $Z_i = X_i$ or $Z_i = Y_i$

Sometimes desired properties are improved.

Difficult to prove.

Example (from Banks, Carson, Nelson, and Nicol):

Set $m = 2^6 = 64$, $a = 13$, $c = 0$. What happens?

i	X_i	X_i	X_i	X_i
0	1	2	3	4
1	13	26	39	52
2	41	18	56	36
3	21	42	63	20
4	17	34	51	**4**
⋮	⋮	⋮	⋮	⋮
8	33	**2**	35	
⋮	⋮	⋮	⋮	
16	**1**		**3**	

The minimum period = 4... terrible random numbers!!!!

Why does cycling occur so soon?

Theorem (Knuth):
The generator $X_{i+1} = aX_i \bmod 2^n$ $(n > 3)$ can have cycle length of at most $2^{n-2}, n > 3$. This is achieved when X_0 is odd and $a = 8k + 3$ or $a = 8k + 5$ for some k. (See last example.)

Theorem (Knuth):
$X_{i+1} = (aX_i + c) \bmod m, c > 0$ has full cycle if
i. c and m are relatively prime
ii. $a - 1$ is a multiple of every prime which divides m
iii. $a - 1$ is a multiple of 4 if 4 divides m.

Corollary:
$X_{i+1} = aX_i \bmod 2^n$ $(c, n > 1)$ has full cycle if c is odd and $a = 4k + 1$ for some k.

Tests for PRN's

We'll look at two classes of tests:

Goodness-of-fit tests — are the PRN's approximately Unif(0,1)?

Independence tests — are the PRN's approximately independent?

If a particular generator passes both types of tests, we'll be happy to use the PRN's it generates.

All tests are of the form H_0 (our null hypothesis) vs. H_1 (the alternative hypothesis).

We regard H_0 as the status quo, so we'll only reject H_0 if we have "ample" evidence against it.

In fact, we want to avoid incorrect rejections of the null hypothesis. Thus, when we design the test, we'll set the "level of significance"

$$\alpha \equiv P(\text{Reject } H_0 | H_0 \text{ true}) = P(\text{Type I error})$$

(typically $\alpha = 0.05$ or 0.1).

1. χ^2 goodness-of-fit.

Test $H_0 : R_1, R_2, \ldots R_n \sim$ Unif(0,1).

Divide the n observations into k cells. If you choose equi-probable cells $[0, 1/k), [1/k, 2/k), \ldots, [(k-1)/k, 1]$, then a particular observation R_j will fall in a particular cell with prob $1/k$.

If $O_i \equiv \#$ of R_j's in cell i, then (since the R_j's are i.i.d.), we can easily see that $O_i \sim \text{Bin}(n, 1/k)$, $i = 1, 2, \ldots, k$.

Thus, the expected number of R_j's to fall in cell i will be $E_i \equiv \text{E}[O_i] = n/k$, $i = 1, 2, \ldots, k$.

We'll reject the null hypothesis H_0 if the O_i's don't match well with the E_i's.

The χ^2 goodness-of-fit statistic is

$$\chi_0^2 \equiv \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i}.$$

A large value of this statistic indicates a bad fit. In fact, we *reject* the null hypothesis H_0 (that the observations are uniform) if $\chi_0^2 > \chi^2_{\alpha,k-1}$, where $\chi^2_{\alpha,k-1}$ is the appropriate $(1-\alpha)$ quantile from a χ^2 table, i.e., $\Pr(\chi^2_{k-1} < \chi^2_{\alpha,k-1}) = 1 - \alpha$.

If $\chi_0^2 \le \chi^2_{\alpha,k-1}$, we *fail to reject* H_0.

Usual recommendation from baby stats class: For χ^2 g-o-f test to work, pick k, n such that $E_i \ge 5$ and n at least 30. But...

Unlike what you learned in baby stats class, when we're testing PRN generators, we usually have a *huge* number of observations n (at least millions) with a large number of cells k. When k is large, can use the following approximation.

$$\chi^2_{\alpha,k-1} \approx (k-1)\left[1 - \frac{2}{9(k-1)} + z_\alpha \sqrt{\frac{2}{9(k-1)}}\right]^3,$$

where z_α is the appropriate standard normal quantile.

Remarks: (1) 16807 PRN generator usually passes the g-o-f test just fine. (2) We'll show how to do g-o-f tests for other distributions later on — just doing uniform PRN's for now. (3) Other g-o-f tests: Kolmogorov-Smirnov test, Anderson-Darling test, etc.

Illustrative Example (BCNN). $n = 100$ observations, $k = 10$ intervals. Thus, $E_i = 10$ for $i = 1, 2, \ldots, 10$. Further, suppose that $O_1 = 13$, $O_2 = 8$, ..., $O_{10} = 11$. (In other words, 13 observations fell in the cell [0,0.1], etc.)

Turns out that

$$\chi_0^2 \equiv \sum_{i=1}^{k} \frac{(O_i - E_i)^2}{E_i} = 3.4.$$

Let's take $\alpha = 0.05$. Then from the back of the book, we have

$$\chi^2_{\alpha,k-1} = \chi^2_{0.05,9} = 15.9.$$

Since $\chi_0^2 < \chi^2_{\alpha,k-1}$, we fail to reject H_0, and so we'll assume that the observations are approximately uniform.

0.34	0.90	0.25	0.89	0.87	0.44	0.12	0.21	0.46	0.67
0.83	0.76	0.79	0.64	0.70	0.81	0.94	0.74	0.22	0.74
0.96	0.99	0.77	0.67	0.56	0.41	0.52	0.73	0.99	0.02
0.47	0.30	0.17	0.82	0.56	0.05	0.45	0.31	0.78	0.05
0.79	0.71	0.23	0.19	0.82	0.93	0.65	0.37	0.39	0.42
0.99	0.17	0.99	0.46	0.05	0.66	0.10	0.42	0.18	0.49
0.37	0.51	0.54	0.01	0.81	0.28	0.69	0.34	0.75	0.49
0.72	0.43	0.56	0.97	0.30	0.94	0.96	0.58	0.73	0.05
0.06	0.39	0.84	0.24	0.40	0.64	0.40	0.19	0.79	0.62
0.18	0.26	0.97	0.88	0.64	0.47	0.60	0.11	0.29	0.78

Table 7.3 Computations for Chi-Square Test

Interval	O_i	E_i	$O_i - E_i$	$(O_i - E_i)^2$	$\frac{(O_i - E_i)^2}{E_i}$
1	8	10	−2	4	0.4
2	8	10	−2	4	0.4
3	10	10	0	0	0.0
4	9	10	−1	1	0.1
5	12	10	2	4	0.4
6	8	10	−2	4	0.4
7	10	10	0	0	0.0
8	14	10	4	16	1.6
9	10	10	0	0	0.0
10	11	10	1	1	0.1
	100	100	0		3.4

2. Tests for Independence.

$H_0 : R_1, R_2, \ldots, R_n$ are independent.

First look at **Runs Tests**.

Consider some examples of coin tossing:

A. H, T, H, T, H, T, H, T, H, T,. . . (negative correlation)

B. H, H, H, H, H, T, T, T, T, T,. . . (positive correlation)

C. H, H, H, T, T, H, T, T, H, T,. . . ("just right")

A *run* is a series of similar observations.

In A above, the runs are: "H", "T", "H", "T",. . . . (many runs)

In B, the runs are: "HHHHH", "TTTTT", ... (very few runs)

In C: "HHH", "TT", "H", "TT",. . . . (medium number of runs)

A runs test will reject the null hypothesis of independence if there are "too many" or "too few" runs, whatever that means.

Runs Test "Up and Down". Consider the following sequence of uniforms.

$$.41 \ .68 \ .89 \ .84 \ .74 \ .91 \ .55 \ .71 \ .36 \ .30 \ .09 \ldots$$

If the uniform increases, put a $+$; if it decreases, put a $-$ (like H's and T's). Get the sequence

$$+ + - - + - + - - - \ldots$$

Here are the associated runs:

$$++, \ --, \ +, \ -, \ +, \ ---, \ \ldots$$

So do we have too many or two few runs?

Let A denote the total number of runs "up and down" out of n observations. ($A = 6$ in the above example.) Obviously, $1 \leq A \leq n - 1$.

Amazing Fact: If n is large (at least 20) and the R_j's are actually independent, then

$$A \approx \text{Nor}\left(\frac{2n-1}{3}, \frac{16n-29}{90}\right).$$

So if you have 100 observations, you might expect around 67 runs!

We'll reject the null hypothesis if A is too big or small. Here's the standardized test statistic: Let

$$Z_0 = \frac{A - \text{E}[A]}{\sqrt{\text{Var}(A)}}.$$

Thus, we reject H_0 if $|Z_0| > z_{\alpha/2}$. E.g., if $\alpha = 0.05$, we reject if $|Z_0| > 1.96$.

Example: Independence

- Consider the following sequence of numbers, read from left to right:

0.12	0.01	0.23	0.28	0.89	0.31	0.64	0.28	0.83	0.93
0.99	0.15	0.33	0.35	0.91	0.41	0.60	0.27	0.75	0.88
0.68	0.49	0.05	0.43	0.95	0.58	0.19	0.36	0.69	0.87

Next Class

- Random-Variate Generation