CHAPTER 7

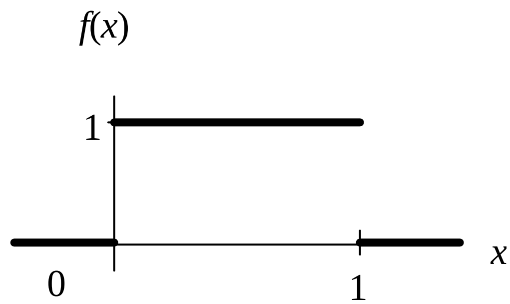
Random-Number Generators

7.1 Introduction

The Goal

All stochastic simulations need to "generate" IID U(0,1) "random numbers" somehow

Density function: $f(x) = \begin{cases} 1 & \text{if } 0 \le x \le 1 \\ 0 & \text{otherwise} \end{cases}$



Reason: Observations on *all* other RVs/processes require U(0,1) input

Early Methods

Physical

- Cast lots
- Dice
- Cards
- Urns
  - Shewhart quality-control methods (“normal bowl”)
  - “Student’s” experiments on distribution of sample correlation coefficient
- Lotteries

Mechanical

- Spinning disks (Kendall/Babington-Smith, 10,000 digits)

Electrical

- ERNIE
- RAND Corp. Tables: *A Million Random Digits with 100,000 Normal Deviates*

Other schemes

- Pick digits “randomly” from Scottish phone directory or census reports
- Decimals in expansion of *p* to 100,000 places

Algorithmic, Sequential Computer Methods

Sequential: the next "random" number is determined by one or several of its predecessors according to a fixed mathematical formula

The *midsquare method*: von Neumann and Metropolis, 1945

Start with Z_0 = 4-digit positive integer

Z_1 = middle 4 digits of Z_0^2 (append 0s if necessary to left of Z_0^2 to get exactly 8 digits); $U_1 = Z_1$, with decimal point at left

Z_2 = middle 4 digits of Z_1^2; $U_2 = Z_2$, with decimal point at left

Z_3 = middle 4 digits of Z_2^2; $U_3 = Z_3$, with decimal point at left. etc.

i	Z_i	U_i	Z_i^2
0	7182	—	51581124
1	5811	0.5811	33767721
2	7677	0.7677	58936329
3	9363	0.9363	87665769
4	6657	0.6657	44315649
5	3156	0.3156	09960336
6	9603	0.9603	92217609
7	2176	0.2176	04734976
8	7349	0.7349	54007801
9	0078	0.0078	00006084
10	0060	0.0060	00003600
11	0036	0.0036	00001296
12	0012	0.0012	00000144
13	0001	0.0001	00000001
14	0000	0.0000	00000000
15	0000	0.0000	00000000
.	.	.	.
.	.	.	.

Other problems with midsquare method:

Not really "random"—entire sequence determined by Z_0

If a Z_i ever reappears, the entire sequence will be recycled

(This *will* occur, since the only choices for 4-digit positive integers are 0000, 0001, 0002, ..., 9999)

"Design" generators so U_i's "appear" to be IID U(0,1) and cycle length is long

<u>Can We Generate "Truly" Random Numbers?</u>

"True" randomness:

Only possible with physical experiment having output ~ U(0,1)

Still some interest in this (counting gamma rays from space)

Problems:

Not reproducible

Impractical for computers (wire in special circuits)

Practical view: produce stream of numbers that *appear* to be IID U(0,1) draws

Use theoretical properties as far as possible

Empirical tests

<u>Criteria for Random-Number Generators</u>

1. "Appear to be distributed uniformly on [0, 1] and independent
2. Fast, low memory
3. Be able to reproduce a particular stream of random numbers. Why?
    a. Makes debugging easier
    b. Use identical random numbers to simulate alternative system configurations for sharper comparison
4. Have provision in the generator for a large number of separate (non-overlapping) *streams* of random numbers; usually such streams are just carefully chosen subsequences of the larger overall sequence

Most RNGs are fast, take very little memory

But beware: There are many RNGs in use (and in software) that have extremely poor statistical properties

7.2 Linear Congruential Generators

Still the most common type (Lehmer, 1954)

Specify four parameters (all nonnegative integers):

Z_0 = *seed* (or starting value)

m = *modulus* (or divisor)

a = *multiplier*

c = *increment*

Then $Z_1, Z_2, Z_3, \ldots$ are recursively generated by $Z_i = (aZ_{i-1} + c) \pmod{m}$, i.e., Z_i is the *remainder* of dividing $aZ_{i-1} + c$ by m.

Thus, $0 \le Z_i \le m - 1$ for each i, so let $U_i = Z_i / m$, so $0 \le U_i < 1$.

Objections to LCGs

Not really "random"—indeed, an explicit formula for each Z_i is

$$Z_i = \left[a^i Z_0 + \frac{c(a^i - 1)}{a - 1} \right] \pmod{m}$$

Cycles when previous Z_i reappears (only m choices, so cycle length is $\le m$)

U_i's can only take on the discrete values $0/m, 1/m, 2/m, \ldots, (m-1)/m$, but they're supposed to be continuous (but pick $m \ge 10^9$ or more in practice)

Example of a "Toy" LCG

$m = 16$, $a = 5$, $c = 3$, $Z_0 = 7$, so formula is $Z_i = (5Z_{i-1} + 3) \pmod{16}$

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

Cycle length (or *period*) here is 16 = m, the longest possible

Questions:

Can the period be "predicted" in advance?

Can a full period be guaranteed?

Full-Period Theorem (Hull and Dobell, 1966)

In general, cycle length determined by parameters m, a, and c:

The LCG $Z_i = (aZ_{i-1} + c) \pmod{m}$ has full period (m) if and only all three of the following hold:

1. c and m are relatively prime (i.e., the only positive integer that divides both c and m is 1).
2. If q is any prime number that divides m, then q also divides $a - 1$.
3. If 4 divides m, then 4 also divides $a - 1$.

Choice of initial seed Z_0 does not enter in

Checking these conditions for a "real" generator may be difficult

Condition 1 says that if $c = 0$, then full period is impossible (since m divides both m and $c = 0$)

Theorem is about period only—says nothing about uniformity of a subcycle, independence, or other statistical properties

Other Desirable Properties Satisfied by LCGs

Fast
Low storage
Reproducible (remember Z_0)
Restart in middle (remember last Z_i, use as Z_0 next time)
Multiple streams (save separated seeds)
Good statistical properties (depends on choice of parameters)

Implementation/Portability Issues

LCGs (and other generators) generally deal with very large integers (like m)

Usually, choose $m \geq 2.1$ billion $\approx 2.1 \times 10^9$

Take care in coding, especially in high-level languages (FORTRAN, C)

Use sophisticated abstract-algebra tricks to "break up" the arithmetic on large integers, then reassemble, to avoid need to store and operate on large integers

Often, use *integer overflow* to effect modulo m division

Suppose $m = 2^b$, where b = number of data bits in a word (often, $b = 31$)

Earlier toy example: $Z_i = (5Z_{i-1} + 3) \pmod{16}$

Suppose on a mythical machine with $b = 4$, so $m = 2^4 = 16$

$Z_6 = 5$, so $Z_7 = (5 \times 5 + 3) \pmod{16} = (28) \pmod{16} = 12$

28 = 11100 in binary

4-bit computer can store only rightmost 4 bits, or 1100 = 12

Thus, division modulo 2^b is *automatic* via integer overflow

Problem: often get poor statistical properties using $m = 2^b$

Solution: *simulated division*:, an algebraic trick to recover most of the computing efficiency of integer overflow with $m \neq 2^b$ (details in text)

Some Specific "Good" (With One Exception) LCGs

Mixed ($c > 0$):

m	a	c
2^{31} = 2,147,483,648	314,159,269	453,806,245
2^{35} = 34,359,738,368	5^{15} = 30,517,578,125	1

Multiplicative ($c = 0$, the case for most LCGs):

m	a	
2^{31} = 2,147,483,648	$2^{16} + 3$ = 65,539	"RANDU," a ***terrible*** generator
$2^{31} - 1$ = 2,147,483,647	7^5 = 16,807 630,360,016 742,938,285 397,204,094	SIMAN, Arena, AweSim SIMSCRIPT, simlib GPSS/H GPSS/PC

RAND

FORTRAN, and C code in the Appendix 7A

$m = 2^{31} - 1 = 2{,}147{,}483{,}647$

$a = 630{,}360{,}016$

$c = 0$ (*multiplicative* LCG, so period *cannot* be full; but period = $m - 1$)

Default seeds for 100 streams spaced 100,000 apart

See comments in code for getting Z_i's, setting seeds

7.3 Other Kinds of Generators

7.3.1 More General Congruences

LCGs are a special case of the form $Z_i = g(Z_{i-1}, Z_{i-2}, ...)$ (mod m), $U_i = Z_i/m$, for some function g

Examples:

$g(Z_{i-1}) = aZ_{i-1} + c$	LCG
$g(Z_{i-1}, Z_{i-2}, ..., Z_{i-q}) = a_1 Z_{i-1} + a_2 Z_{i-2} + ... + a_q Z_{i-q}$	multiple recursive generator
$g(Z_{i-1}) = a' Z_{i-1}^2 + aZ_{i-1} + c$	quadratic CG
$g(Z_{i-1}, Z_{i-2}) = Z_{i-1} + Z_{i-2}$	Fibonacci (bad)

7.3.2 Composite Generators

Combine two (or more) individual generators in some way

Shuffling

Fill a vector of length 128 (say) from generator 1

Use generator 2 to pick one of the 128 in the vector

Fill the hole with the next value from generator 1, use generator 2 to pick one of the 128 in the vector, etc.

Evidence: shuffling a bad generator improves it, but shuffling a good generator doesn't gain much.

Differencing LCGs

Z_{1i} and Z_{2i} from LCGs with different moduli

Let $Z_i = (Z_{1i} - Z_{2i})$ (mod m); $U_i = Z_i / m$

Very long period (like 10^{18}); very good statistical properties

Very portable (micros, different languages)

Wichmann/Hill

Use three LCGs to get U_{1i}, U_{2i}, and U_{3i} sequences

Let U_i = fractional part of $U_{1i} + U_{2i} + U_{3i}$

Long period, good statistics, portability

But .... later shown to be equivalent to a LCG (!!!)

Combined Multiple Recursive Generators

Recall the single MRG: $Z_i = (a_1 Z_{i-1} + a_2 Z_{i-2} + \dots + a_q Z_{i-q}) \pmod{m}$, $U_i = Z_i/m$

Have J MRGs running simultaneously: $\{Z_{1,i}\}$, $\{Z_{2,i}\}$, ..., $\{Z_{J,i}\}$

Let m_1 be the modulus used in the first of these J MRGs

For constants d_1, d_2, ..., d_J, define

$$Y_i = (d_1 Z_{1,i} + d_2 Z_{2,i} + \dots + d_J Z_{J,i}) \pmod{m_1}$$

Return $U_i = Y_i / m$

Must choose constants carefully, but extremely long periods and extremely good statistical behavior can be achieved

Specific example in text for $J = 2$, $q = 3$ for both MRGs, $d_1 = 1$, $d_2 = -1$, with small, fast, portable C code in Appendix 7B, with 10,000 streams spaced 10^{16} apart

Period is approximately 3.1×10^{57}

Excellent statistical properties through 32 dimensions (see Sec. 7.4.2)

7.3.3 Tausworthe and Related Generators

Tausworthe Generators

Originated in cryptography

Generate sequence of bits b_1, b_2, b_3, ... via congruence

$$b_i = (b_{i-r} + b_{i-q}) \pmod 2) = \begin{cases} 0 \text{ if } b_{i-r} = b_{i-q} \\ 1 \text{ if } b_{i-r} \neq b_{i-q} \end{cases}$$

Various algorithms to group bits into U_i's

Can achieve very long periods

Theoretical appeal: for properly chosen parameters, can prove that over a cycle,

- mean $\approx 1/2$ (as for true U(0,1))
- variance $\approx 1/12$ (as for true U(0,1))
- autocorrelation ≈ 0 (as for true IID sequence)
- n-tuples ~ $U(0,1)^n$ (a problem with LCGs)

"Unpredictable" Generators (Blum/Blum/Shub)

Another way to generate a sequence of bits

Pick p and q to be large (like 40-digit) prime numbers, set $m = pq$

Generate $X_i = X_{i-1}^2 \pmod m$

Let b_i = *parity* of X_i (0 if even, 1 if odd), also equal to rightmost bit of X_i

Result: Discovering nonrandomness in bit sequence is computationally equivalent to factoring m into the product of p times q (which is widely believed to be essentially impossible in any reasonable time period)

Thus, the sequence is really "random" in any practical sense

New York Times (Tuesday April 19, 1988, Section C): "The Quest for True Randomness Finally Appears Successful"

7.4 Testing Random-Number Generators

Since RNGs are completely deterministic, we need to test them to see if they appear to be random and IID uniform on [0, 1]

General advice: be wary of "canned" RNGs in software that is not specifically simulation software, especially if they are not thoroughly documented; perhaps even test them before using

Two types of tests: Empirical and Theoretical

7.4.1 Empirical Tests

Use trial generator to generate some U's, apply statistical tests

Give information only on that part of a generator's cycle examined (local) — is this good or bad?

Some examples:

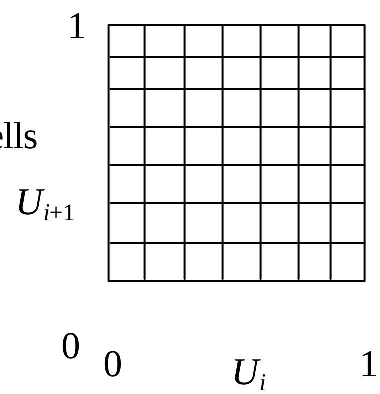
Chi-square, K-S tests for U(0,1) (all parameters known)

Serial test—generalize chi-square test to higher dimensions

Two dimensions—subdivide unit square into subcells

Test statistic = $\sum_{all\ cells} \frac{(Observed - Expected)^2}{Expected}$

Provides indirect test of independence



Runs tests—direct test of independence

"Run up of length i" occurs if exactly i U's in a row go up

Under H_0: independence, know P(Run up of length i), any i

Observe frequency of runs up, compare with probabilities

Get a chi-squared test statistic

Direct test for correlation—see text for details

7.4.2 Theoretical Tests

Don't generate any U's, but use analytical properties of generator

Not statistical "tests" at all

Apply to full period of a generator (global—relevance??)

Full-Period Values

LCGs, Tausworthe generators

Prove "sample" mean of U's over a full period is $1/2 - 1/(2m)$

Prove "sample" variance of U's over a full period is $1/12 - 1/(12m^2)$

Lattice Structure of LCGs (and Other Kinds of Generators)

"Random Numbers Lie Mainly in the Planes" (G. Marsaglia, 1968)

Think of generating all pairs (U_i, U_{i+1}), i over a full period

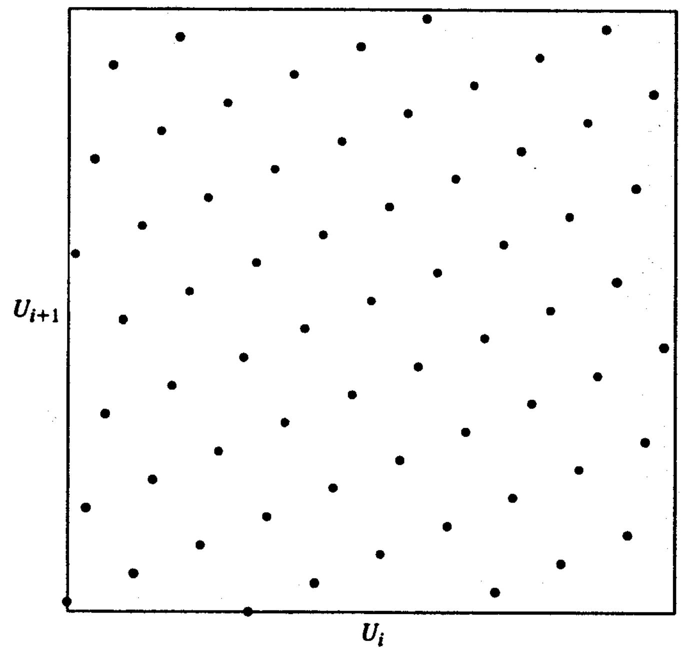
Think of generating all triples (U_i, U_{i+1}, U_{i+2}), i over a full period, etc.

The generated d-tuples all lie on parallel $d - 1$ dimensional hyperplanes cutting through the d-dimensional unit cube

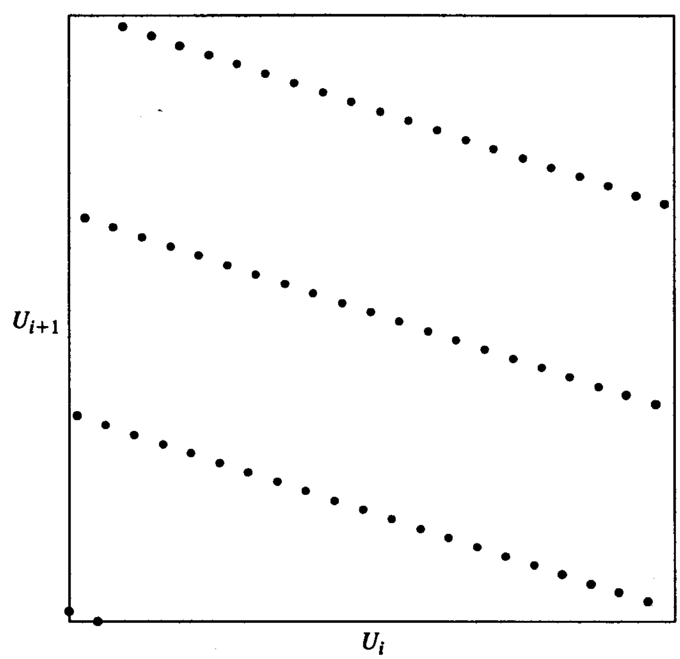
Spectral and *lattice* tests—measure spacing of hyperplanes (smaller is better)

Two-dimensional

m = 64, a = 37, c = 1:

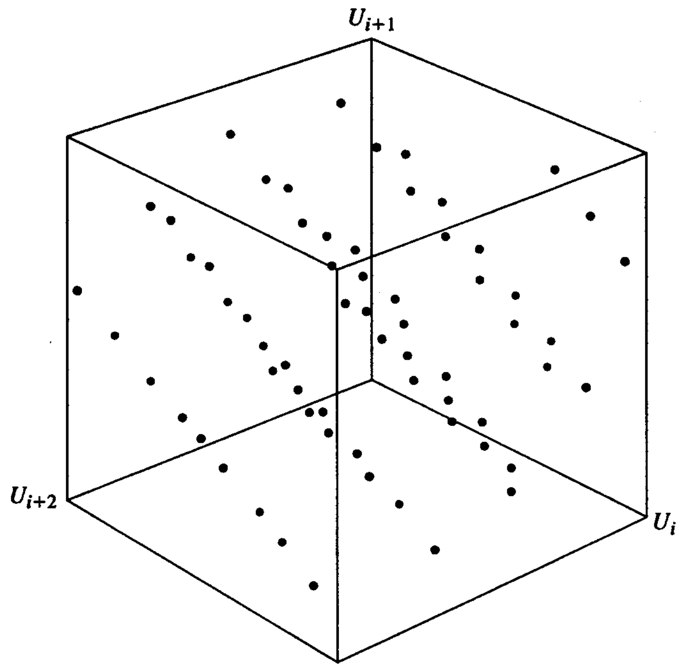


m = 64, a = 21, c = 1 (only change from above: a = 21 rather than 37):

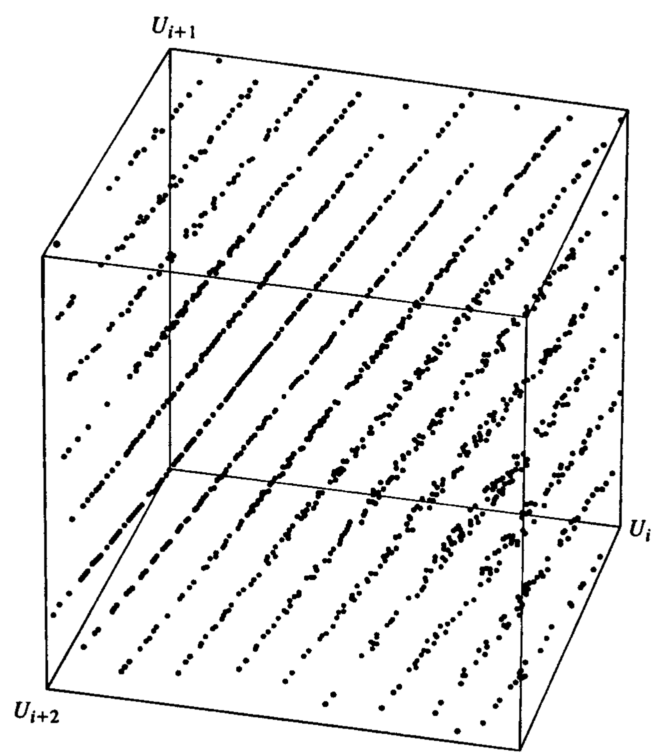


Three-dimensional

$m = 64$, $a = 37$, $c = 1$:



$m = 2^{31} = 2{,}147{,}483{,}648$, $a = 2^{16} + 3 = 65{,}539$, $c = 0$ (RANDU):



7.4.3 Some General Observations on Testing

Beware of "canned" generators—especially in non-simulation software, and especially if poorly documented

Insist on documentation of the generator—check with "tried and true" ones

Don't "seed" the generator with the square root of the clock or any other such scatterbrained scheme—we *want* to get reproducibility of the random-number stream

Reasonably safe idea:

Use one of the generators from Appendix 7A (if period of 10^9 is long enough ... which may or may not be true) or 7B (if longer period is needed)

Highly portable

Provide default streams with controlled, known spacing

Take care to avoid overlapping the streams