ISYE 6644 Week 8
Random Number Generator
Note

1. **The key in generating random numbers**

- Uniform (0,1) random numbers are the key to random variate generation in simulation - you transform uniforms to get other RVs.

- The goal: give an algorithm that produces a sequence of pseudo-random number (PRNs) R1, R2…that "appears" to be i.i.d Unif(0,1)

Desired properties of algorithm

- output appears to be i.i.d. Unif(0,1)
- very fast
- ability to reproduce any sequence it generates

2. **Some Lousy Generators / poor generators**

- Random devices (con: storage and repeat)
- Random number tables
- Mid-Square Method (J von Neumann): the method is terrible, there is positive correlation in Ri's , and occasionally degenerates (consider Xi = 0003)
- Fibonacci and Additive Congruential generator: (small numbers follow small numbers)

These methods are also no good!!

$$X_i = (X_{i-1} + X_{i-2}) \bmod m, \quad i = 1, 2, \ldots,$$

where $R_i = X_i/m$, m is the *modulus*, X_{-1}, X_0 are *seeds*, and $a = b \bmod m$ iff a is the remainder of b/m, e.g., $6 = 13 \bmod 7$.

3. **Linear Congruential Generator:**

LCGs are the most widely used generators. These are pretty good when implemented properly.

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

$R_i = X_i/m$, $i = 1, 2, \ldots$

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.

- Desert island generator

Better Example (desert island generator): Here's our old 16807 implementation (BFS 1987), which I've translated from FORTRAN. It works fine, is fast, and is full-period with cycle length > 2 billion,

$$X_i = 16807\, X_{i-1} \bmod (2^{31} - 1).$$

Algorithm: Let X_0 be an integer seed between 1 and $2^{31} - 1$.

For $i = 1, 2, \ldots,$

$K \leftarrow \lfloor X_{i-1} / 127773 \rfloor$ (integer division leaves no remainder)
$X_i \leftarrow 16807(X_{i-1} - 127773K) - 2836K$
if $X_i < 0$, then set $X_i \leftarrow X_i + 2147483647$
$R_i \leftarrow X_i * 4.656612875\text{E-}10$

4. Tausworthe Generator

Tausworthe Generator

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

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

where $c_j = 0$ or 1. Looks a bit like a generalization of LCGs.

Usual implementation (saves computational effort):

$$B_i = (B_{i-r} + B_{i-q}) \bmod 2 = B_{i-r} \text{ XOR } B_{i-q} \quad (0 < r < q).$$

Obtain

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

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

The period of 0-1 bits is always $2^q - 1 = 31$.

Question 6 1 pts

(Lesson 6.4: Tausworthe Generators.) Suppose that a Tausworthe generator gave you the series of bits 1010101. If you use all 7 bits, what Unif(0,1) random number would that translate to?

- a. 0.3825
- b. 0.5
- c. 0.6641
- d. 0.9826

5. **Generalizations of LCGs**

A Simple Generalization:

$X_i = \left(\sum_{j=1}^{q} a_i X_{i-j}\right) \bmod m$, where the a_i's are constants.

Extremely large periods possible (up to $m^q - 1$ if parameters are chosen properly). But watch out! — Fibonacci is a special case.

Combinations of Generators:

Can combine two generators $X_1, X_2, \ldots$ and $Y_1, Y_2, \ldots$ to construct $Z_1, Z_2, \ldots$. Some suggestions:

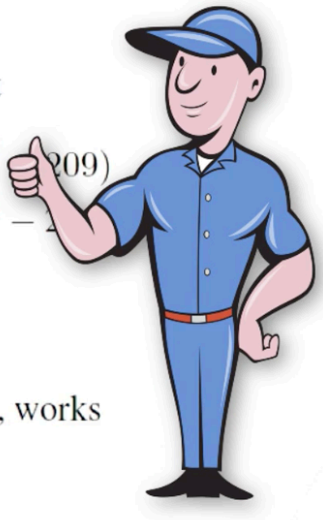
- Set $Z_i = (X_i + Y_i) \bmod m$
- Shuffling
- Set $Z_i = X_i$ or $Z_i = Y_i$

A Really Good Combined Generator due to L'Ecuyer (1999) (and discussed in Law 2015).

Initialize $X_{1,0}, X_{1,1}, X_{1,2}, X_{2,0}, X_{2,1}, X_{2,2}$. For $i \geq 3$, set

$$
\begin{aligned}
X_{1,i} &= (1{,}403{,}580\, X_{1,i-2} - 810{,}728\, X_{1,i-3}) \bmod (2^{32} - 209) \\
X_{2,i} &= (527{,}612\, X_{2,i-1} - 1{,}370{,}589\, X_{2,i-3}) \bmod (2^{32} - \\
Y_i &= (X_{1,i} - X_{2,i}) \bmod (2^{32} - 209) \\
R_i &= Y_i/(2^{32} - 209)
\end{aligned}
$$

As crazy as this generator looks, it's actually pretty simple, works well, and has an amazing cycle length of about 2^{191}!



- Mersenne Twister has period of 2^19937-1 (a prime number)

6. How to choose a good generator - Theory

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

Lots of these types of cycle length results.

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

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

Theorem: The *multiplicative* generator $X_i = aX_{i-1} \bmod m$, with prime m has full period $(m - 1)$ if and only if (i) m divides $a^{m-1} - 1$; and (ii) for all integers $i < m - 1$, m does not divide $a^i - 1$.

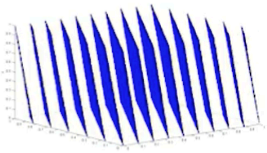
Remark: For $m = 2^{31} - 1$, it can be shown that 534,600,000 multipliers yield full period, the "best" of which is $a = 950{,}706{,}376$ (Fishman and Moore 1986).

Geometric Considerations

Theorem: The k-tuples $(R_i, \ldots, R_{i+k-1})$, $i \geq 1$, from multiplicative generators lie on parallel hyperplanes in $[0, 1]^k$.

The following geometric quantities are of interest.

- Minimum number of hyperplanes (in all directions). Find the multiplier that maximizes this number.
- Maximum distance between parallel hyperplanes. Find the multiplier that minimizes this number.
- Minimum Euclidean distance between adjacent k-tuples. Find the multiplier that maximizes this number.



Remark: The RANDU generator is particularly bad since it lies on only 15 hyperplanes.

Can also look at one-step serial correlation.

Serial Correlation of LCGs (Greenberger 1961):

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

This upper bound is very small for m in the range of 2 billion and, say, $a = 16807$.

Lots of other theory considerations that can be used to evaluate the performance of a particular PRN generator.

7. How to choose a good generator - Statistics Test

- **Goodness-of-fit Test**
- We will look at Goodness-of-fit tests (Unif(0,1) ?)and Independence tests
- Level of significance = Type I error = Prob(reject H_0 / H_0 is true)

χ^2 Goodness-of-Fit Test

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

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

Tally how many of the n observations fall into the k cells. 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 $O_i \sim \text{Bin}(n, \frac{1}{k})$, $i = 1, 2, \ldots, k$.

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

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 $\chi_0^2 > \chi_{\alpha,k-1}^2$, where $\chi_{\alpha,k-1}^2$ is the appropriate $(1-\alpha)$ quantile from a χ^2 table, i.e., $P(\chi_{k-1}^2 < \chi_{\alpha,k-1}^2) = 1 - \alpha$.

Unlike what you learned in baby stats class, when we test 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, we can use the 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 PRNs for now. (3) Other g-o-f tests: Kolmogorov–Smirnov test, Anderson–Darling test, etc.

- **Independence Test**
- Run tests

Definition: 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. There are various types of runs tests; we'll discuss two of them.

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

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

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

We'll reject H_0 if A is too big or small. The test statistic is

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

and we reject H_0 if $|Z_0| > z_{\alpha/2}$.

- Runs test "above and below the mean"

1. **The key in generating random numbers**

- Uniform (0,1) random numbers are the key to random variate generation in simulation - you transform uniforms to get other RVs.

- The goal: give an algorithm that produces a sequence of pseudo-random number (PRNs) R1, R2…that "appears" to be i.i.d Unif(0,1)

Desired properties of algorithm

- output appears to be i.i.d. Unif(0,1)
- very fast
- ability to reproduce any sequence it generates

2. **Some Lousy Generators / poor generators**

- Random devices (con: storage and repeat)
- Random number tables
- Mid-Square Method (J von Neumann): the method is terrible, there is positive correlation in Ri's , and occasionally degenerates (consider Xi = 0003)
- Fibonacci and Additive Congruential generator: (small numbers follow small numbers)

These methods are also no good!!

$$X_i = (X_{i-1} + X_{i-2}) \bmod m, \quad i = 1, 2, \ldots,$$

where $R_i = X_i/m$, m is the *modulus*, X_{-1}, X_0 are *seeds*, and $a = b \bmod m$ iff a is the remainder of b/m, e.g., $6 = 13 \bmod 7$.

3. **Linear Congruential Generator:**

LCGs are the most widely used generators. These are pretty good when implemented properly.

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

$R_i = X_i/m,\ i = 1, 2, \ldots$

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.

- Desert island generator

Better Example (desert island generator): Here's our old 16807 implementation (BFS 1987), which I've translated from FORTRAN. It works fine, is fast, and is full-period with cycle length > 2 billion,

$$X_i = 16807\, X_{i-1} \bmod (2^{31} - 1).$$

Algorithm: Let X_0 be an integer seed between 1 and $2^{31} - 1$.

For $i = 1, 2, \ldots,$

$K \leftarrow \lfloor X_{i-1} / 127773 \rfloor$ (integer division leaves no remainder)
$X_i \leftarrow 16807(X_{i-1} - 127773K) - 2836K$
if $X_i < 0$, then set $X_i \leftarrow X_i + 2147483647$
$R_i \leftarrow X_i * 4.656612875\text{E-}10$

4. Tausworthe Generator

Tausworthe Generator

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

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

where $c_j = 0$ or 1. Looks a bit like a generalization of LCGs.

Usual implementation (saves computational effort):

$$B_i = (B_{i-r} + B_{i-q}) \bmod 2 = B_{i-r} \text{ XOR } B_{i-q} \quad (0 < r < q).$$

Obtain

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

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

The period of 0-1 bits is always $2^q - 1 = 31$.

Question 6 — 1 pts

(Lesson 6.4: Tausworthe Generators.) Suppose that a Tausworthe generator gave you the series of bits 1010101. If you use all 7 bits, what Unif(0,1) random number would that translate to?

- a. 0.3825
- b. 0.5
- c. 0.6641
- d. 0.9826

5. **Generalizations of LCGs**

A Simple Generalization:

$X_i = \left(\sum_{j=1}^{q} a_i X_{i-j}\right) \bmod m$, where the a_i's are constants.

Extremely large periods possible (up to $m^q - 1$ if parameters are chosen properly). But watch out! — Fibonacci is a special case.

Combinations of Generators:

Can combine two generators $X_1, X_2, \ldots$ and $Y_1, Y_2, \ldots$ to construct $Z_1, Z_2, \ldots$. Some suggestions:

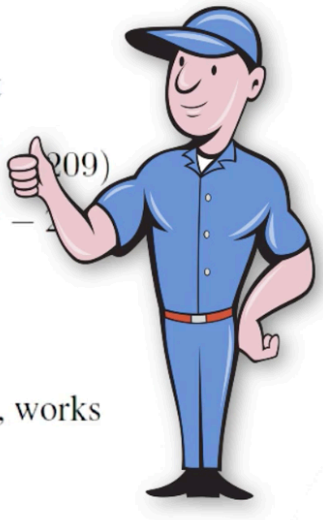
- Set $Z_i = (X_i + Y_i) \bmod m$
- Shuffling
- Set $Z_i = X_i$ or $Z_i = Y_i$

A Really Good Combined Generator due to L'Ecuyer (1999) (and discussed in Law 2015).

Initialize $X_{1,0}, X_{1,1}, X_{1,2}, X_{2,0}, X_{2,1}, X_{2,2}$. For $i \geq 3$, set

$$
\begin{aligned}
X_{1,i} &= (1{,}403{,}580\, X_{1,i-2} - 810{,}728\, X_{1,i-3}) \bmod (2^{32} - 209) \\
X_{2,i} &= (527{,}612\, X_{2,i-1} - 1{,}370{,}589\, X_{2,i-3}) \bmod (2^{32} - \\
Y_i &= (X_{1,i} - X_{2,i}) \bmod (2^{32} - 209) \\
R_i &= Y_i/(2^{32} - 209)
\end{aligned}
$$

As crazy as this generator looks, it's actually pretty simple, works well, and has an amazing cycle length of about 2^{191}!



- Mersenne Twister has period of 2^19937-1 (a prime number)

6. How to choose a good generator - Theory

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

Lots of these types of cycle length results.

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

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

Theorem: The *multiplicative* generator $X_i = aX_{i-1} \bmod m$, with prime m has full period $(m - 1)$ if and only if (i) m divides $a^{m-1} - 1$; and (ii) for all integers $i < m - 1$, m does not divide $a^i - 1$.

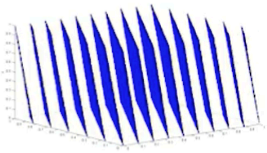
Remark: For $m = 2^{31} - 1$, it can be shown that 534,600,000 multipliers yield full period, the "best" of which is $a = 950{,}706{,}376$ (Fishman and Moore 1986).

Geometric Considerations

Theorem: The k-tuples $(R_i, \ldots, R_{i+k-1})$, $i \geq 1$, from multiplicative generators lie on parallel hyperplanes in $[0, 1]^k$.

The following geometric quantities are of interest.

- Minimum number of hyperplanes (in all directions). Find the multiplier that maximizes this number.
- Maximum distance between parallel hyperplanes. Find the multiplier that minimizes this number.
- Minimum Euclidean distance between adjacent k-tuples. Find the multiplier that maximizes this number.



Remark: The RANDU generator is particularly bad since it lies on only 15 hyperplanes.

Can also look at one-step serial correlation.

Serial Correlation of LCGs (Greenberger 1961):

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

This upper bound is very small for m in the range of 2 billion and, say, $a = 16807$.

Lots of other theory considerations that can be used to evaluate the performance of a particular PRN generator.

7. How to choose a good generator - Statistics Test

- **Goodness-of-fit Test**
- We will look at Goodness-of-fit tests (Unif(0,1) ?)and Independence tests
- Level of significance = Type I error = Prob(reject H_0 / H_0 is true)

χ^2 Goodness-of-Fit Test

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

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

Tally how many of the n observations fall into the k cells. 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 $O_i \sim \text{Bin}(n, \frac{1}{k})$, $i = 1, 2, \ldots, k$.

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

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 $\chi_0^2 > \chi_{\alpha,k-1}^2$, where $\chi_{\alpha,k-1}^2$ is the appropriate $(1-\alpha)$ quantile from a χ^2 table, i.e., $P(\chi_{k-1}^2 < \chi_{\alpha,k-1}^2) = 1 - \alpha$.

Unlike what you learned in baby stats class, when we test 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, we can use the 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 PRNs for now. (3) Other g-o-f tests: Kolmogorov–Smirnov test, Anderson–Darling test, etc.

- **Independence Test**
- Run tests

Definition: 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. There are various types of runs tests; we'll discuss two of them.

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

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

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

We'll reject H_0 if A is too big or small. The test statistic is

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

and we reject H_0 if $|Z_0| > z_{\alpha/2}$.

OT

- Runs test "above and below the mean"