Discrete-Event Simulation:

A First Course

Section 2.2: Lehmer Random Number Generators: Implementation

0-13-142917-5

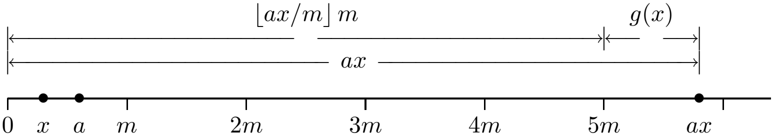
Section 2.2: Lehmer Random Number Generators: Implementation

- For 32-bit systems, $2^{31} - 1$ is the largest prime
- We will develop an $m = 2^{31} - 1$ Lehmer generator
  - portable, efficient
  - in ANSI C
- ANSI C standard:

$$\begin{aligned} \texttt{LONG\_MAX} &\geq 2^{31} - 1 \\ \texttt{LONG\_MIN} &\leq -(2^{31} - 1) \end{aligned}$$

Overflow Is Possible

- Recall that $g(x) = ax \bmod m$
- The ax product can be as big as $a(m-1)$



  0-13-142917-5

Implementation

- If integers $> m$ cannot be represented, integer overflow is possible
- Not possible to evaluate $g(x)$ in "obvious" way
- Example 2.2.1: Consider $(a, m) = (48271, 2^{31} - 1)$
  - $a(m-1) \simeq 1.47 \times 2^{46} \Longrightarrow$ at least 47 bits
  - However, $ax \bmod m$ no more than 31 bits
- Consider $(a, m) = (7, 13)$ from Example 2.1.1 for a 5-bit machine

$$a(m-1) = 84 \simeq 1.31 \times 2^{6} \Longrightarrow \text{at least 7 bits}$$

Type Considerations

- Why `long`?
  - ANSI C standard guarantees 32 bits for `long`
  - Most contemporary computers are 32-bit
- Why not `float` or `double`?
  - Floating-point representation is inexact
  - An efficient integer-based implementation exists
- Why not `long long` — guarantees 64 bits?

  Requires overhead on 32-bit systems
- 64-bit machines will not alleviate the problem

  m would be $2^{64} - 59$, overflow still possible

Algorithm Development

- Want an integer-based implementation
- No calculation can give result $> m = 2^{31} - 1$
- If m was not prime, then $m = aq$

$$g(x) = ax \bmod m = \cdots = a(x \bmod q)$$

Note: mod before multiply!

- But m is prime, so $m = aq + r$ where

$$q = \left\lfloor \frac{m}{a} \right\rfloor \qquad\qquad r = m \bmod a$$

Want remainder smaller than quotient ($r < q$)

Example 2.2.4: (q, r) Decomposition of m

- Consider $(a, m) = (48\,271, 2^{31} - 1)$

$$q = \left\lfloor \frac{m}{a} \right\rfloor = 44\,488 \qquad\qquad r = m \bmod a = 3399$$

- Consider $(a, m) = (16\,807, 2^{31} - 1)$

$$q = 127\,773 \qquad\qquad r = 2836$$

- Note that $r < q$ in both cases
- This (modulus compatibility) is important later!

Rewriting $g(x)$ To Avoid Overflow

$$
\begin{aligned}
g(x) &= ax \bmod m \\
&= ax - m\lfloor ax/m \rfloor \\
&= ax + \Big[- m\lfloor x/q \rfloor + m\lfloor x/q \rfloor \Big] - m\lfloor ax/m \rfloor \\
&= \Big[ax - (aq + r)\lfloor x/q \rfloor \Big] + \Big[m\lfloor x/q \rfloor - m\lfloor ax/m \rfloor \Big] \\
&= \Big[a\big(x - q\lfloor x/q \rfloor\big) - r\lfloor x/q \rfloor \Big] + \Big[m\lfloor x/q \rfloor - m\lfloor ax/m \rfloor \Big] \\
&= \Big[a(x \bmod q) - r\lfloor x/q \rfloor \Big] + m\Big[\lfloor x/q \rfloor - \lfloor ax/m \rfloor \Big] \\
&= \gamma(x) + m\delta(x)
\end{aligned}
$$

- Mods are done before multiplications!

Theorem 2.2.1: $\delta(x)$ Is Either 0 Or 1

Theorem (2.2.1)

If $m = aq + r$ *is prime and* $r < q$ *and* $x \in \mathcal{X}_m$

$$\delta(x) = 0 \qquad or \qquad \delta(x) = 1$$

where $\delta(x) = \lfloor x/q \rfloor - \lfloor ax/m \rfloor$

Proof.

Note for $u, v \in \mathbb{R}$ with $0 < u - v < 1$, $\lfloor u \rfloor - \lfloor v \rfloor$ is 0 or 1

Consider

$$\frac{x}{q} - \frac{ax}{m} = x\left(\frac{1}{q} - \frac{a}{m}\right) = x\left(\frac{m-aq}{mq}\right) = \frac{xr}{mq}$$

and since $r < q$

$$0 < \frac{xr}{mq} < \frac{x}{m} \leq \frac{m-1}{m} < 1$$

$\square$

Theorem 2.2.1: $\delta(x)$ Depends Only On $\gamma(x)$

Theorem (2.2.1)

With $\gamma(x) = a(x \bmod q) - r\lfloor x/q \rfloor$

$$\begin{aligned} \delta(x) = 0 \quad &\textit{iff.} \quad \gamma(x) \in \mathcal{X}_m \\ \delta(x) = 1 \quad &\textit{iff.} \quad -\gamma(x) \in \mathcal{X}_m \end{aligned}$$

Proof.

- If $\delta(x) = 0$, then $g(x) = \gamma(x) + m\delta(x) = \gamma(x) \in \mathcal{X}_m$
  If $\gamma(x) \in \mathcal{X}_m$, then $\delta(x) \neq 1$ otherwise $g(x) \notin \mathcal{X}_m$
- If $\delta(x) = 1$, then $-\gamma(x) \in \mathcal{X}_m$ otherwise $g(x) = \gamma(x) + m \notin \mathcal{X}_m$
  If $-\gamma(x) \in \mathcal{X}_m$, then $\delta(x) \neq 0$ otherwise $g(x) \notin \mathcal{X}_m$

$\square$

Algorithm 2.2.1: Computing $g(x)$

- Evaluates $g(x) = ax \bmod m$ with no values $> m - 1$

Algorithm 2.2.1

```
t = a * (x % q) - r * (x / q);      /* t = γ(x) */
if (t > 0)
    return (t);                      /* δ(x) = 0 */
else
    return (t + m);                  /* δ(x) = 1 */
```

- Returns $g(x) = \gamma(x) + m\delta(x)$
- The ax product is "trapped" in $\delta(x)$
- No overflow

Modulus Compatibility

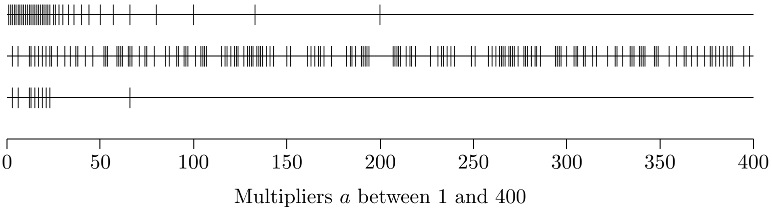
- We must have $r < q$ in $m = aq + r$ (see proof of Theorem 2.2.1)
- Multiplier a is *modulus-compatible* with m iff. $r < q$
- Here, choose a modulus-compatible with $m = 2^{31} - 1$

  Then algorithm 2.2.1 can port to any 32-bit machine
- E.g., $a = 48271$ is modulus-compatible with $m = 2^{31} - 1$

$$r = 3399 \qquad\qquad q = 44\,488$$

Modulus-Compatible and Full-Period

- No modulus-compatible multipliers $> (m-1)/2$
- More densely distributed on low end
- Consider (tiny) modulus $m = 401$: (Row 1: MP, Row 2: FP, Row 3: MP & FP)



Modulus-Compatibility and Smallness

- Multiplier a is "small" iff. $a^2 < m$
- If a is small, then a is modulus-compatible

  All multipliers from 1 to $\lfloor\sqrt{m}\rfloor = 46340$ *are modulus-compatible*

- If a is modulus-compatible, a is not necessarily small

  $a = 48271$ *is modulus-compatible with* $2^{31} - 1$ *but is not small*

- Start with a small (therefore modulus-compatible) multiplier

  Search until the first full-period multiplier is found (Alg. 2.1.1)

Algorithm 2.2.2: Generating All Full-Period Modulus-Compatible Multipliers

- Find one full-period modulus-compatible (FPMC) multiplier
- The following (an extension of Alg. 2.1.2) generates all others

Algorithm 2.2.1

```
i = 1;
x = a;
while (x != 1) {
    if ((m % x < m / x) and (gcd(i, m - 1) == 1))
        /* x is full-period & modulus-compatible */
    i++;
    x = g(x); /* use Alg. 2.2.1 to evaluate g(x) */
}
```

Example 2.2.6: FPMC Multipliers For $m = 2^{31} - 1$

- For $m = 2^{31} - 1$ and FPMC $a = 7$, there are 23093 FPMC multipliers

$$
\begin{aligned}
7^{1} \bmod 2147483647 &= 7 \\
7^{5} \bmod 2147483647 &= 16807 \\
7^{113039} \bmod 2147483647 &= 41214 \\
7^{188509} \bmod 2147483647 &= 25697 \\
7^{536035} \bmod 2147483647 &= 63295 \\
&\vdots
\end{aligned}
$$

- $a = 16807$ is a "minimal" standard
- $a = 48271$ provides (slightly) more random sequences

Randomness

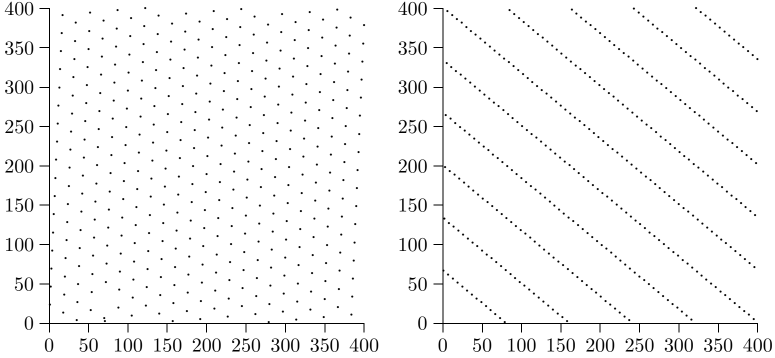
- Choose the FPMC multiplier that gives "most random" sequence
- No universal definition of randomness
- In 2-space, $(x_0, x_1), (x_1, x_2), (x_2, x_3), \ldots$ form a *lattice* structure
- For any integer $k \geq 2$, the points

$$(x_0, x_1, \ldots, x_{k-1}), (x_1, x_2, \ldots, x_k), (x_2, x_3, \ldots, x_{k+1}), \ldots$$

  form a lattice structure in k-space
- Numerically analyze *uniformity* of the lattice

  E.g., Knuth's spectral test

Random Numbers Falling In The Planes



$(a, m) = (23, 401)$ $\qquad$ $(a, m) = (66, 401)$

ANSI C Implementation

- A Lehmer RNG in ANSI C with $(a, m) = (48271, 2^{31} - 1)$:

Random Method

```
Random(void) {
    static long state = 1;

    const long A = 48271; /* multiplier */
    const long M = 2147483647; /* modulus */
    const long Q = M / A; /* quotient */
    const long R = M % A; /* remainder */
        long t = A * (state % Q) - R * (state / Q);
    if (t > 0)
        state = t;
    else
        state = t + M;
    return ((double) state / M);
}
```

A Not-As-Good RNG Library

- ANSI C library `<stdlib.h>` provides the function `rand()`
- Simulates drawing from $0, 1, 2, \ldots, m-1$ with $m \geq 2^{15} - 1$
- Value returned is not normalized; typical to use

```
u = (double) rand() / RAND_MAX;
```

- ANSI C standard does *not* specify algorithm details
- For scientific work, avoid using `rand()` (Summit, 1995)

A Good RNG Library

- Defined in the source files `rng.h` and `rng.c`
- Based on the implementation considered in this lecture
  - `double Random(void)`
  - `void PutSeed(long seed)`
  - `void GetSeed(long *seed)`
  - `void TestRandom(void)`
- Initial seed can be set directly, via prompt, or by system clock
- `PutSeed()` and `GetSeed()` often used together
- $a = 48271$ is the default multiplier

Example 2.2.10: Using the RNG

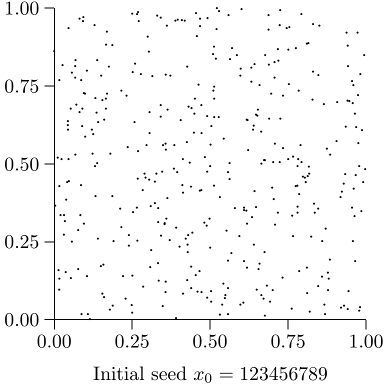
- The following generates 400 2-space points at random

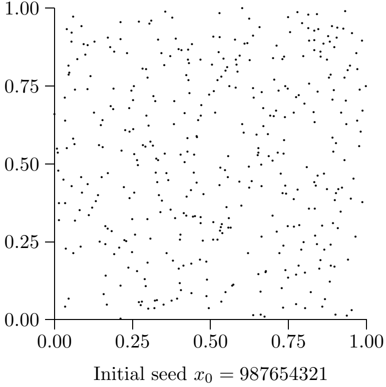
Generating 2-Space Points

```
seed = 123456789;      /* or 987654321 */
PutSeed(seed);
x0 = Random();
for (i = 0; i < 400; i++) {
    x[i+1] = Random();
    Plot(x[i], x[i+1]);      /* graphics function */
}
```

- Generate one sequence with each initial seed

Scatter Plot Of 400 Pairs





Observations on Randomness

- In previous figure, no lattice structure is evident
- Appearance of randomness is an illusion
- If all $m - 1 = 2^{31} - 2$ points were generated, lattice would be evident
- Herein lies distinction between *ideal* and *good* RNGs

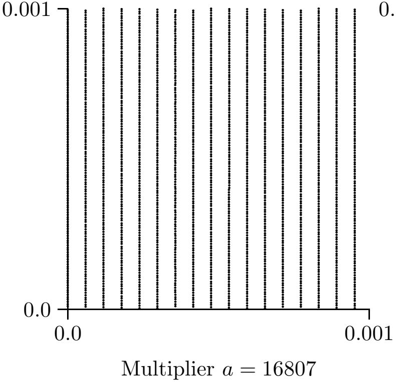
  0-13-142917-5

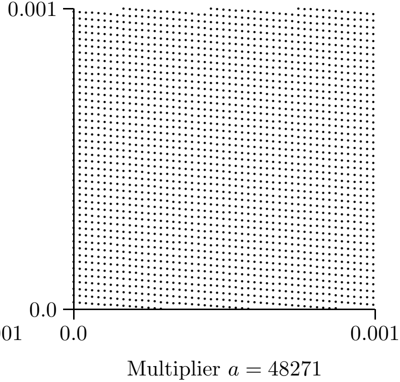
Example 2.2.11

- Plotting all pairs (x_i, x_{i+1}) for $m = 2^{31} - 1$ would give a black square
- Any tiny square should appear (approximately) the same
- "Zoom in" to square with corners $(0, 0)$ and $(0.001, 0.001)$

  `seed = 123456789; PutSeed(seed);` x_0 `= Random();` `for (`i `= 0;` i `< 2147483646;` i`++) {` x_{i+1} `= Random();` `if ((`$x_i < 0.001$`) and (`$x_{i+1} < 0.001$`))` `Plot(`x_i`,` x_{i+1}`); }`
- Results for multipliers $a = 16807$ and $a = 48271$ on the next slide

Scatter Plots for $m = 2^{31} - 1$





- Further justification for using $a = 48271$ over $a = 16807$

Other Multipliers and Considerations

- for $m = 2^{31} - 1$ there are 534 600 000 multipliers a that are full period
- 23 903 of these are modulus compatible
- Section 10.1 discusses statistical tests for these numbers, but a lot of research has already been done
- Nonrepresentative Subsequences: What if only 20 random numbers were needed and you chose seed $x_0 = 109\,869\,724$?
- Resulting 20 random numbers:

0.64	0.72	0.77	0.93	0.82	0.88	0.67	0.76	0.84	0.84
0.74	0.76	0.80	0.75	0.63	0.94	0.86	0.63	0.78	0.67

Fast CPUs and cycling

- How long does it take to generate a full period for $m = 2^{31} - 1$?
  - 1980's : days
  - 1990's : hours
  - Today : minutes
  - Soon : seconds
- Recall:
  - *Ideal* generator draws from an urn "with replacement"
  - *Our* generator draws from an urn "without replacement"
  - Distinction is irrelevant *if number of draws is small* compared to m
- *Cycling*: generating more than $m - 1$ random values
- Cycling must be avoided within a single simulation