Random Number Generation and Monte Carlo Simulation

Lawrence M. Leemis and Stephen K. Park, Discrete-Event Simul A First Course, Prentice Hall, 2006

Hui Chen

Department of Mathematics and Computer Science
Virginia State University
Petersburg, Virginia

January 29, 2015

Need for Random Number Generators

- Single Server Queue and Simple Inventory System
- Two trace-driven simulation programs: *ssq1* and *sis1*
- The usefulness of these programs depends on the availability of the traces
  - What if more data is needed?
  - What if the input data set is small or unavailable?
  - What if the model changes?
- A random number generator addresses all the problems
  - It produces random real values between 0.0 and 1.0
  - The output can be converted to *random variate* via mathematical transformations

Random Number Generators (RNG)

- Types of generators
  - Table look-up generators
  - Hardware generators
  - Algorithmic (software) generators
- Desired criteria
  - Randomness: output passes all reasonable statistical tests of randomness
  - Controllability: able to reproduce output, if desired
  - Portability: able to produce the same output on a wide variety of computer systems
  - Efficiency: fast, minimal computer resource requirements
  - Documentation: theoretically analyzed and extensively tested
- Algorithmic generators meet the above criteria and are widely accepted

Algorithmic Generators

- An *ideal* RNG produces output such that each value in the interval $0.0 < u < 1.0$ is equally likely to occur
- A *good* RNG produces output that is *almost* statistically indistinguishable from an ideal RNG
- We will construct a good RNG satisfying all our criteria
  - Lehmer Random Number Generators

Lehmer Random Number Generators: Conceptual Model

- Conceptual Model
  - Choose a large positive integer m. This defines the set $\mathcal{X}_m = \{1, 2, \ldots, m-1\}$
  - Fill a (conceptual) urn with the elements of $\mathcal{X}_m$
  - Each time a random number u is needed, draw an integer x at "random" from the urn and let $u = x/m$
- Each draw simulates a sample of an independent identically distributed sequence of $\mathit{Uniform}(0, 1)$
- The possible values are $1/m, 2/m, \ldots (m-1)/m$.
- It is important that m be large so that the possible values are densely distributed between 0.0 and 1.0
- Practical and special consideration
  - 0.0 and 1.0 are impossible: for avoiding problems associated with certain random-variate-generation algorithms
  - Although we would like to draw from the urn with replacement, we will draw without replacement for practical reasons: if m is large and the number of draws is small relative to m, the distinctino is largely irrelevant

Lehmer's Algorithm for Random Number Generation

- Lehmer Generator: the integer sequence $x_0, x_1, \ldots \in \mathcal{X}_m$ is defined by the iterative equation

$$x_{i+1} = g(x_i) = ax_i \mod m \tag{1}$$

  where
  - $\mathcal{X}_m = \{1, 2, \ldots, m-1\}$
  - $x_0 \in \mathcal{X}_m$ is called the *initial seed*.
  - modulus m is a fixed large *prime* integer
  - multiplier $a \in \mathcal{X}_m$

Lehmer Generators: a, x_0 and m

- $0 \leq g(x) < m$
- 0 must not occur since $g(0) = a \cdot 0 \mod m = 0 \mod m = 0$
- Since m is prime, $g(x) \neq 0$ if $x \in \mathcal{X}_m$
- If $x_0 \in \mathcal{X}_m$, then $x_i \in \mathcal{X}_m$ for all $i \geq 0$.

Pseudo-random Number Generators

- If the multiplier and prime modulus are chosen properly, a Lehmer generator is statistically indistinguishable from drawing from $\mathcal{X}_m$ with replacement.
- Note that there is *nothing* random about a Lehmer generator
  - For this reason, it is called a *pseudo-random number generator*

Intuitive Explanation

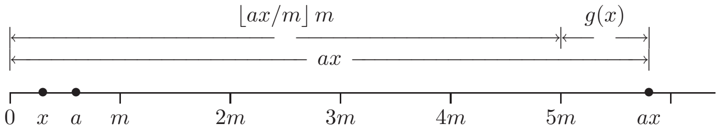


Figure : Leher generator geometry

- When ax is divided by m, the remainder is "likely" to be any value between 0 and $m - 1$
- Similar to buying numerous identical items at a grocery store with only dollar bills.
  - a is the price of an item, x is the number of items, and $m = 100$.
  - The change is likely to be any value between 0 and 99 cents.

Parameter Consideration

- The choice of m is dictated, in part, by system considerations
  - In general, we want to choose m to be the largest representable prime integer
  - On a system with 32-bit 2's complement integer arithmetic, $2^{31} - 1$ is a natural choice since it is a prime integer and the largest possible positive integer
  - With 16-bit or 64-bit integer representation, the choice is not obvious, since neither $2^{15} - 1$ nor $2^{63} - 1$ is a prime integer
- Given m, the choice of a must be made with great care (see Example 2.1.1)

Example 2.1.1

- If $m = 13$ and $a = 6$ with $x_0 = 1$ then the sequence is

$$1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1, \ldots$$

where the ellipses (i.e., $\ldots$) indicate the sequence is periodic

- If $m = 13$ and $a = 7$ with $x_0 = 1$ then the sequence is

$$1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1, \ldots$$

Because of the $12, 6, 3$ and $8, 4, 2, 1$ patterns, this sequence appears "less random"

- If $m = 13$ and $a = 5$ then

$$1, 5, 12, 8, 1, \ldots \text{ or } 2, 10, 11, 3, 2, \ldots \text{ or } 4, 7, 9, 6, 4, \ldots$$

This less-than-full-period behavior is obviously undesirable

Central Issues

- For a chosen (a, m) pair, does the function $g(\cdot)$ generate a full-period sequence?
- If a full period sequence is generated, how random does the sequence appear to be?
- Can $ax \mod m$ be evaluated efficiently and correctly?
  - Integer overflow can occur when computing ax

Full Period Considerations

- $b \mod a = b - \lfloor b/a \rfloor a$
- There exists a non-negative integer $c_i = \lfloor ax_i/m \rfloor$ such that

$$x_{i+1} = g(x_i) = ax_i \mod m = ax_i - mc_i$$

Therefore, by induction, we have

$$\begin{aligned}
x_1 &= ax_0 - mc_0 \\
x_2 &= ax_1 - mc_1 = a^2x_0 - m(ac_0 + c_1) \\
x_3 &= ax_2 - mc_2 = a^3x_0 - m(a^2c_0 + ac_1 + c_2) \\
&\vdots \\
x_i &= ax_{i-1} - mc_{i-1} = a^ix_0 - m(a^{i-1}c_0 + a^{i-2}c_1 + \ldots + c_{i-1})
\end{aligned}$$

Full Period Consideration

- Since $x_i \in \mathcal{X}_m$, we have $x_i = x_i \mod m$. Therefore, letting $c = a^{i-1}c_0 + a^{i-2}c_1 + \ldots + c_{i-1}$, we have

$$x_i = a^i x_0 - mc = (a^i x_0 - mc) \mod m = a^i x_0 \mod m$$

Theorem 2.1.1

If the sequence $x_0, x_1, x_2, \ldots$ is produced by a Lehmer generator with multiplier a and modulus m then

$$x_i = a^i x_0 \mod m$$

- It is an eminently bad idea to compute x_i by first computing a_i
- Theorem 2.1.1 has significant theoretical value

Full Period Consideration

- Since $(b_1 b_2 \dots b_n) \mod a = (b_1 \mod a)(b_2 \mod a) \dots (b_n \mod a) \mod a$, we have

$$x_i = a^i x_0 \mod m = (a^i \mod m) x_0 \mod m$$

- Fermat's little theorem states that if p is a prime which does not divide a, then $a^{p-1} \mod p = 1$. Then,

$$x_{m-1} = (a^{m-1} \mod m) x_0 \mod m = x_0$$

Theorem 2.1.2

if $x_0 \in \mathcal{X}_m$ and the sequence $x_0, x_1, x_2, \dots$ is produced by a Lehmer generator with multiplier a and prime modulus m then there is a positive integer p with $p \leq m-1$ such that $x_0, x_1, x_2, \dots x_{p-1}$ are all different and

$$x_{i+p} = x_i \qquad i = 0, 1, 2, \dots$$

That is, the sequence is periodic with fundamental period p. In addition, $(m-1) \mod p = 0$.

Full Period Consideration

- If we pick any initial seed $x_0 \in \mathcal{X}_m$ and generate the sequence $x_0, x_1, x_2, \ldots$ then x_0 will occur again
- Further x_0 will reappear at index p that is either $m - 1$ or a divisor of $m - 1$
- The pattern will repeat forever
- We are interested in choosing *full-period multipliers* where $p = m - 1$

Example 2.1.2

- Full-period multipliers generate a virtual circular list with $m - 1$ distinct elements.

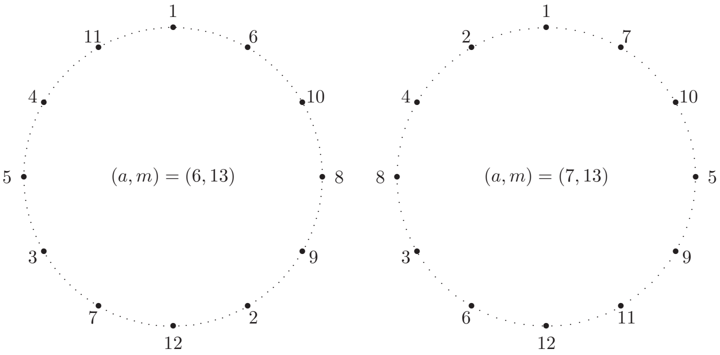


Figure : Two full-period generators.

Finding Full Period Multipliers

Algorithm 2.1.1

```
p = 1;
x = a;
while (x != 1) {
    p ++;
    x = (a * x) % m;
}
if (p == m - 1)
    /* a is a full-period multiplier */
else
    /* a is not a full-period multiplier */
```

- This algorithm is a slow-but-sure way to test for a full-period multiplier

Frequency of Full-Period Multipliers

- Given a prime modulus m, how many corresponding full-period multipliers are there?

Theorem 2.1.3

If m is prime and $p_1, p_2, \ldots, p_r$ are the (unique) prime factors of $m-1$ then the number of full-period multipliers in $\mathcal{X}_m$ is

$$\frac{(p_1-1)(p_2-1)\ldots(p_r-1)}{p_1 p_2 \ldots p_r}(m-1)$$

- Example 2.13 If $m = 13$ then $m - 1 = 12 = 2^2 \cdot 3$. Therefore, there are $\frac{(2-1)(3-1)}{2 \cdot 3}(13-1) = 4$ full-period multipliers (i.e., 2, 6, 7, and 11)

Example 2.1.4

- If $m = 2^{31} - 1 = 2147483647$ then since the prime decomposition of $m - 1$ is

$$m - 1 = 2^{31} - 2 = 2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331$$

the number of full-period multipliers is

$$\left(\frac{1 \cdot 2 \cdot 6 \cdot 10 \cdot 30 \cdot 150 \cdot 330}{2 \cdot 3 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331}\right)\left(2 \cdot 3^2 \cdot 7 \cdot 11 \cdot 31 \cdot 151 \cdot 331\right) = 534600000$$

- Therefore, approximately 25% of the multipliers are full-period

Finding All Full-Period Multipliers

- Once one full-period multiplier has been found, then all others can be found in $\mathcal{O}(m)$ time

Algorithm 2.1.2

```
i = 1;
x = a;
while (x != 1) {
    if (gcd(i, m - 1) == 1)
        /* a^i mod m is a full-period multiplier */
    i ++;
    x = (a * x) % m; /* be aware a*x overflow */
}
```

Finding All Full-Period Multipliers

Theorem 2.1.4

If a is any full-period multiplier relative to the prime modulus m then each of the integers

$$a^i \mod m \in \mathcal{X}_m \qquad i = 1, 2, 3, \ldots, m-1$$

is also a full-period multiplier relative to m if and only if i and $m-1$ are relatively prime

Example 2.1.5

- If $m = 13$ then we know from Example 2.1.3 there are 4 full period multipliers. From Example 2.1.1 $a = 6$ is one. Then, since 1, 5, 7, and 11 are relatively prime to 13

$$\begin{aligned} 6^1 \mod 13 &= 6 \\ 6^5 \mod 13 &= 2 \\ 6^7 \mod 13 &= 7 \\ 6^{11} \mod 13 &= 11 \end{aligned}$$

- Equivalently, if we knew $a = 2$ is a full-period multiplier

$$\begin{aligned} 2^1 \mod 13 &= 2 \\ 2^5 \mod 13 &= 6 \\ 2^7 \mod 13 &= 11 \\ 2^{11} \mod 13 &= 7 \end{aligned}$$

Example 2.1.6

- If $m = 2^{31} - 1$ then from Example 2.1.4 there are 534600000 integers relatively prime to $m - 1$. The first few are $i = 1, 5, 13, 17, 19$. $a = 7$ is a full-period multiplier relative to m and therefore

$$\begin{aligned}
7^1 &\mod 2147483647 = 7 \\
7^5 &\mod 2147483647 = 16807 \\
7^{13} &\mod 2147483647 = 252246292 \\
7^{17} &\mod 2147483647 = 52958638 \\
7^{19} &\mod 2147483647 = 447489615
\end{aligned}$$

are full-period multipliers relative to m

Implementation Objective

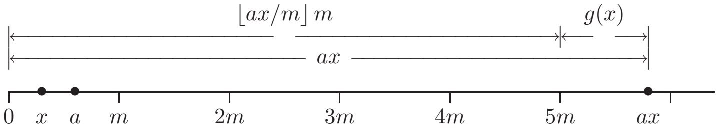
- For 32-bit systems, $2^{31} - 1$ is the largest prime
- We will develop an $m = 2^{31} - 1$ Lehmer generator
  - Portable and efficient
  - in ANSI C
- ANSI C Standard:

$$LONG\_MAX \geq 2^{31} - 1$$

$$LONG\_MIN \leq -(2^{31} - 1)$$

Overflow Is Possible

- Recall that $g(x) = ax \bmod m$
- The ax product can be as big as $a(m-1)$
- If integers $> m$ cannot be represted, integer overflow is possible
- Not possible to evaluate $g(x)$ in "obvious" way



Figure

Example 2.2.1

- Consider $(a, m) = (48271, 2^{31} - 1)$
  - $a(m-1) \simeq 1.47 \times 2^{46} \Rightarrow$ at least 47 bits
  - However, $ax \mod m$ no more than 31 bits
- Consider $(a, m) = (7, 13)$ from Example 2.1.1 for a 5-bit machine
  - $a(m-1) = 84 \simeq 1.31 \times 2^{6} \Rightarrow$ at least 7 bits

Data Type Consideration

- 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 were not prime, then $m = aq$

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

Note: mod before multiply!

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

$$a = \lfloor \frac{m}{a} \rfloor \qquad r = m \mod a$$

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

Example 2.2.4: (q, r) Decomposition of m

- Consider $(a, m) = (48271, 2^{31} - 1)$

$$q = \lfloor \frac{m}{a} \rfloor = 44488 \qquad r = m \mod a = 3399$$

- Consider $(a, m) = (16807, 2^{31} - 1)$

$$q = 127773 \qquad r = 2836$$

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

Rewriting $g(x)$ To Avoid Overflow

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

Mods are done before multiplications!

$\delta(x)$ Is Either 0 Or 1

Theorem 2.2.1 – Part 1

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

$$\delta(x) = 0 \qquad \text{or} \qquad \delta(x) = 1$$

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

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

and since $r < q$

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

□

$\delta(x)$ Depends Only On $\gamma(x)$

Theorem 2.2.1 – Part 2

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

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

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 $\gamma(x) \neq 1$ 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 $g(x) \notin \mathcal{X}_m$

$\square$

Computing $g(x)$

- Evaluates $g(x) = ax \mod m$ 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 proudct 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
- Example: $a = 48271$ is modulus-compatible with $m = 2^{31} - 1$

$$r = 3399 \qquad\qquad q = 44488$$

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)

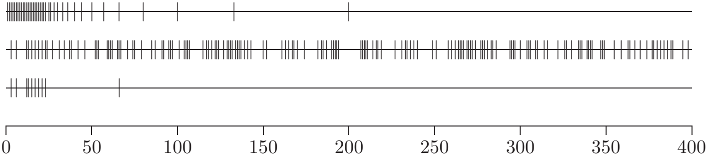


Figure : Modulus-compatible full-period multipliers for $m = 401$

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

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

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

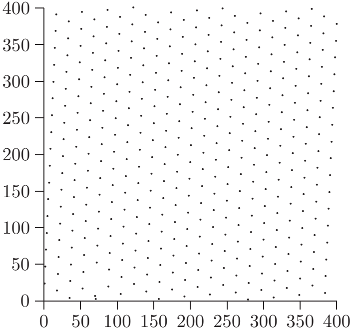
Randomness

- 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 $k > 2$, the points

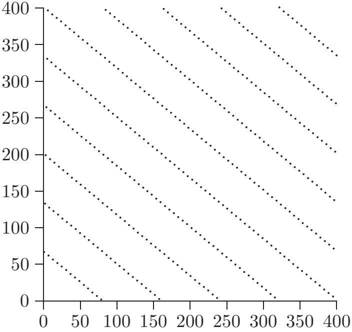
$$(x_0, x_1, \ldots, x_{k-1}), (x_1, x_2, \ldots, x_k), (x_2, x_3, \ldots, x_{k+1}), \ldots$$

  form a lattice structure in k-space
- Numerically analyze uniformity of the lattice
  - Example: Knuth's spectral test

Random Numbers Falling In The Planes



$(a, m) = (23, 401)$



$(a, m) = (66, 401)$

ANSI C Implementation

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

```
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;
    reutrn ((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 $m = 2^{15} - 1$
- 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

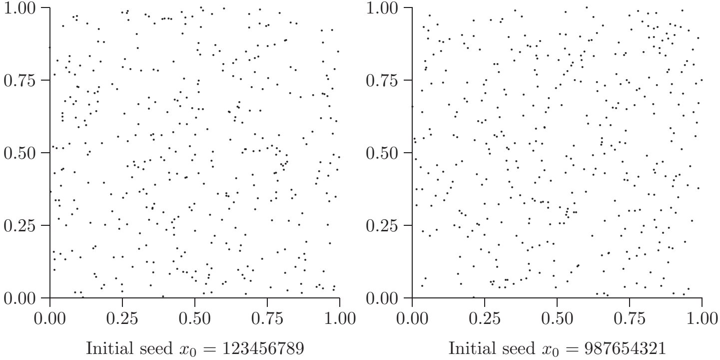
Example 2.2.10: Using the RNG

Generating 2-Space Points

seed = 123456789;
PutSeed(seed);
x_0 = Random();
for (i = 0; i ¡ 400; i++) {
    x_{i+1} = Random();
    Plot(x_i, x_{i+1});
}

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

Example 2.2.11

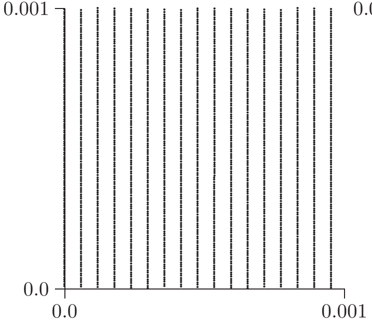
- Plotting all pairs (x_i, x_{i+1}) for $m = 2^{31} - 1$ would give a black square
- Any tiny square should appear (approximately) the same
- "Zoom in" to square with corners $(0, 0)$ and $(0.001, 0.001)$

Generating 2-Space Points and "Zoom in"

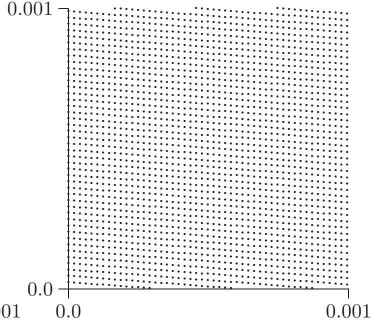
seed = 123456789;
PutSeed(seed);
x_0 = Random();
for (i = 0; i ¡ 2147483646; i++) {
    x_{i+1} = Random();
    if (($x_i < 0.001$) and ($x_{i+1} < 0.001$)) Plot(x_i, x_{i+1});
}

- Results for multipliers $a = 16807$ and $a = 48271$ on the next slide

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



Multiplier $a = 16807$



Multiplier $a = 48271$

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

Other Multipliers and Considerations

- for $m = 2^{31} - 1$ there are 534600000 multipliers a that are full period
- 23903 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 $x_0 = 109869724$?
- 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

Monte Carlo Simulation

- With Empirical Probability, we perform an experiment many times n and count the number of occurrences n_a of an event $\mathcal{A}$
  - The relative frequency of occurrence of event $\mathcal{A}$ is n_a/n
  - The frequency theory of probability asserts that the relative frequency converges as $n \to \infty$

$$Pr(\mathcal{A}) = \lim_{n \to \infty} \frac{n_a}{n}$$

- *Axiomatic Probability* is a formal, set-theoretic approach
  - Mathematically construct the sample space and calculate the number of events $\mathcal{A}$
  - The two are complementary!

Example 2.3.1

- Roll two dice and observe the up faces

(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)
(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6)
(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)
(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	(4, 6)
(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)	(5, 6)
(6, 1)	(6, 2)	(6, 3)	(6, 4)	(6, 5)	(6, 6)

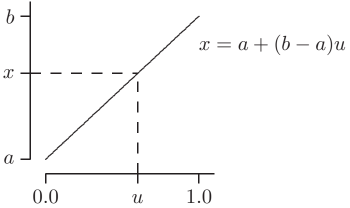
- If the two up faces are summed, an integer-valued random variable, say X, is defined with possible values 2 through 12 inclusive

sum, x:	2	3	4	5	6	7	8	9	10	11	12
$Pr(X = x)$:	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

- $Pr(X = 7)$ could be estimated by replicating the experiment many times and calculating the relative frequency of occurrence of 7's

Random Variates

- A Random Variate is an algorithmically generated realization of a random variable
- $u = Random()$ generates a $Uniform(0,1)$ random variate
- How can we generate a $Uniform(a,b)$ variate?



Generating a Uniform Random Variate

```
double Uniform(double a, double b)        /* use a < b */ {
  return (a + (b - a) * Random());
}
```

Equilikely Random Variates

- *Uniform*$(0,1)$ random variates can also be used to generate an *Equilikely*(a,b) random variate

$$
\begin{aligned}
0 < u < 1 &\iff 0 < (b-a+1)u < b-a+1 \\
&\iff 0 \le \lfloor (b-a+1)u \rfloor \le b-a \\
&\iff a \le a + \lfloor b-a+1)u \rfloor \le b \\
&\iff a \le x \le b
\end{aligned}
$$

- Specifically, $x = a + \lfloor (b-a+1)u \rfloor$

Generating an Equilikely Random Variate

```
long Equilikely(long a, long b)       /* use a < b */ {
    return (a + (long)((b - a + 1) * Random()));
}
```

Examples

- **Example 2.3.3** To generate a random variate x that simulates rolling two fair dice and summing the resulting up faces, use

$$x = Equilikely(1, 6) + Equilikely(1, 6);$$

Note that this is *note* equivalent to

$$x = Equilikely(2, 12);$$

- **Example 2.3.4** To select an element x at random from the array a[0], a[1], . . . , a[n-1] use

$$i = Equilikely(0, n - 1); x = a[i];$$

Galileo's Dice

- If three fair dice are rolled, which sum is more likely, a 9 or a 10?
  - There are $6^3 = 216$ possible outcomes

$$Pr(X = 9) = \frac{25}{216} \cong 0.116 \qquad \textit{and} \qquad Pr(X = 10) = \frac{27}{216} = 0.125$$

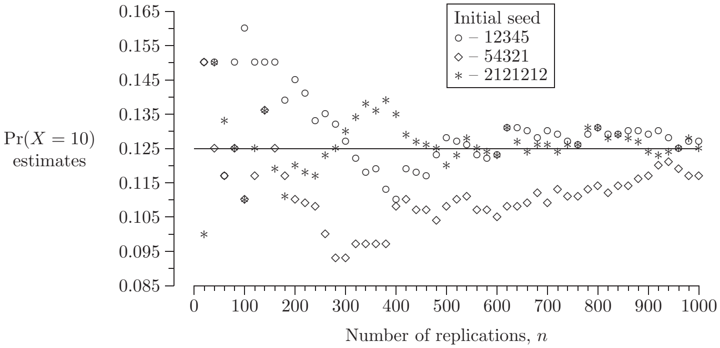
- Program *galileo* calculates the probability of each possible sum between 3 and 18
- The drawback of Monte Carlo simulation is that it only produces an estimate
  - Larger n does not guarantee a more accurate estimate

In-Class Exercise L4-2: Varitions of Galileo's Dice

- Run the Galileo's Dice program (in Blackboard) following the following guideline: seeds.
  - Choose three different seeds
  - Use the number of replications as 20, 40, 100, 200, 400, 1000, 10000, and 100000
  - Show the result in a graph similar to next slide
  - Submit a screen shot showing that you successfully run the program and the Excel workbook or the result from other graphing tools under "In-Class Exercise L4-2"

Example 2.3.6

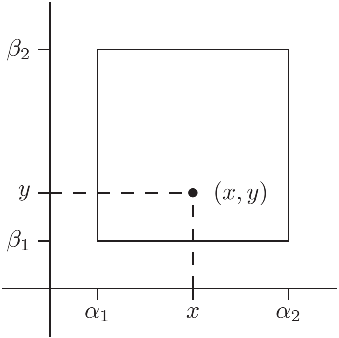
- Frequency probability estimates converge slowly and somewhat erratically



- You should always run a Monte Carlo simulation with multiple initial seeds

Geometric Applications: Rectangle

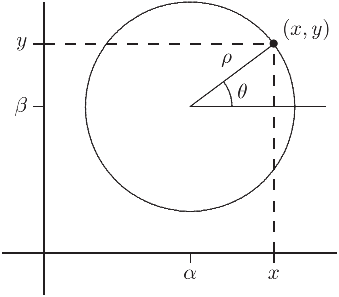
- Generate a point at random inside a rectangle with opposite corners at (α_1, β_1) and (α_2, β_2)



$$x = Uniform(\alpha_1, \alpha_2); \qquad y = Uniform(\beta_1, \beta_2);$$

Geometric Applications: Circle

- Generate a point (x, y) at random on the circumference of a circle with radius ρ and center (α, β)



$$\theta = Uniform(-\pi, \pi); \quad x = \alpha + \rho * cos(\theta); \quad y = \beta + \rho * sin(\theta);$$

Example 2.3.8

- Generate a point (x, y) at random interior to the circle of radius ρ centered at (α, β)

$$\theta = Uniform(-\pi, \pi); \quad r = Uniform(0, \rho);$$

$$x = \alpha + \rho * cos(\theta); \quad y = \beta + r * sin(\theta);$$

Correct?

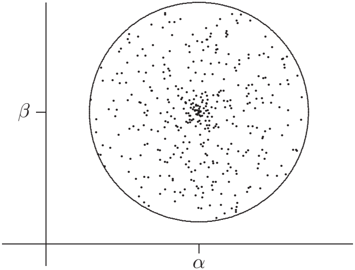
Example 2.3.8

- Generate a point (x, y) at random interior to the circle of radius ρ centered at (α, β)

$$\theta = Uniform(-\pi, \pi); \quad r = Uniform(0, \rho);$$

$$x = \alpha + \rho * cos(\theta); \quad y = \beta + r * sin(\theta);$$

Correct? INCORRECT!

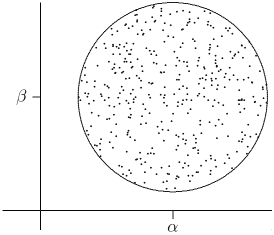


Acceptance/Rejection

- Generate a point at random within a circumscribed square and then either accept or reject the point

Generate a Random Point Interior to a Circle

do {
$x = Uniform(-\rho, \rho)$;
$y = Uniform(-\rho, \rho)$; } while $(x * x + y * y >= \rho * \rho)$;
$x = \alpha + x$; $y = \beta + y$;
return (x, y);

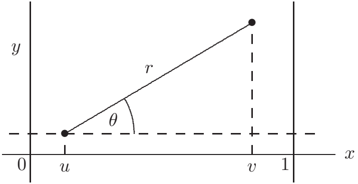


In-Class Exercise L4-3: Geometric Application

- Objective: visually examine correctness of a simulation
- Write a program that randomly generate 1000 points within a rectangle using the method in slide 60 and graph the result
- Write a program that reproduces the incorrect (slide 61) and correct (slide 63 generation of points interior to a circle as shown previous slides.
- Submit the programs and the graphing results (e.g., Excel Workbooks) in Blackboard under "In-Class Exercise L4-3"

Buffon's Needle Problem

- Suppose that an infinite family of infinitely long vertical lines are spaced one unit apart in the (x, y) plane. If a needle of length $r > 0$ is dropped at random onto the plane, what is the probability that it will land crossing at least one line?



  - u is the x-coordinate of the left-hand endpoint
  - v is the x-coordinate of the right-hand endpoint,

$$v = u + rcos\theta$$

  - The needle crosses at least one line if and only if $v > 1$

Program buffon

- Program buffon is a Monte Carlo simulation
  - The random number library can be used to automatically generate an initial seed

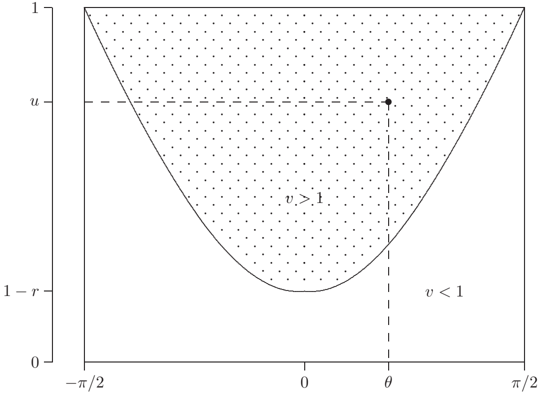
Random Seeding

```
PutSeed(-1);            /* any negative integer will do */
GetSeed(&seed);         /* trap the value of the initial seed */
:
printf("with an initial seed of %ld", seed);
```

- Inspection of the program buffon illustrates how to solve the problem axiomatically

Axiomatic Approach to Buffon's Needle

- "Dropped at random" is interpreted (modeled) to mean that u and θ are independent *Uniform*$(0, 1)$ and *Uniform*$(-\pi/2, \pi/2)$ random variables



Axiomatic Approach to Buffon's Needle

- The shaded region has a curved boundary defined by the equation $u = 1 - rcos\theta$
- if $0 < r \leq 1$, the area of the shaded region is

$$\pi - \int_{-\pi/2}^{\pi/2} (1 - rcos\theta) d\theta = r \int_{-\pi/2}^{\pi/2} cos\theta d\theta = \ldots = 2r$$

- Therefore, because the area of the rectangle is π the probability that the needle will cross at least one line is $2r/\pi$

In-Class Exercise L4-4: Buffon's Needle

- Objective: Compare simulation and axiomatic results (does your simulation program need a test case?)
- Calculate the probability that it will land crossing at least one line for the Buffon's needle problem using the axiomatic result 68.
- Revise the program buffon to output the estimated probability with at least 6 digits after the decimal point.
- Run the revised program buffon for 100, 1000, 10000, 100000, 1000000 replications with 3 different seeds for each number of replications
- Choose appropriate graphs to graph the following,
  - The results from the simulations
  - The axiomatic result
  - The different between the simulations and the axiomatic result (i.e., error)
- Submit the work in Blackboard (a screen shot show the simulation program is running correctly, the revised program, and the graphing result) in Blackboard under "In-Class L4-4"

Axiomatic and Experimental Approaches

- *Axiomatic* and *experimental* approaches are complementary
- Slight changes in assumptions can sink an axiomatic solution
- An axiomatic solution is intractable in some other cases
- Monte Carlo simulation can be used as an alterative in either case
- Four more examples of Monte Carlo simulation
  - Metrics and determinants
  - Craps
  - Hatchek girl
  - Stochastic activity network

Example 1: Matrix and Determinants

- *Matrix:* set of real or complex numbers in a rectangular array
- for matrix A, a_{ij} is the element in row i, column j

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

  where A is $m \times n$, i.e., m rows and n columns
- Interesting quantities: eigenvalue, trace, rank, and determinant

Determinants

- The *determinant* of a 2×2 matrix A is

$$|A| = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}a_{12}$$

- The determinant of a 3×3 matrix A is

$$|A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11} \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12} \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13} \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

Random Matrices

- *Random matrix*: matrix whose elements are random variables
- Consider a 3×3 matrix whose elements are random with positive diagonal, negative off-diagonal elements
- Question: What is the probability the determinant is positive?

$$\begin{vmatrix} +u_{11} & -u_{12} & -u_{13} \\ -u_{21} & +u_{22} & -u_{23} \\ -u_{31} & -u_{32} & +u_{33} \end{vmatrix} > 0$$

- Axiomatic solution is not easily calculated

Specification Model

- Let event $\mathcal{A}$ be that the determinant is positive
- Generate N 3×3 matrices with random elements
- Compute the determinant for each matrix
- Let $n_a =$ number of matrices with determinant > 0
- Probability of interest: $Pr(\mathcal{A}) \cong N_a/N$

Computational Model: Program det

det

```
for (i = 0; i < N; i++) {
    for (j = 1; j <= 3; j++) {
        for (k = 1; k <= 3; k++) {
            a[j][k] = Random();
            if (j != k)
            a[j][k] = -a[j][k];
        }
    }
    temp1 = a[2][2] * a[3][3] - a[3][2] * a[2][3];
    temp2 = a[2][1] * a[3][3] - a[3][1] * a[2][3];
    temp3 = a[2][1] * a[3][2] - a[3][1] * a[2][2];
    x = a[1][1]*temp1 - a[1][2]*temp2 + a[1][3]*temp3;
    if (x > 0)
        count++;
}
printf("%11.9f", (double)count/N);
```

Output From det

- Want N sufficiently large for a good point estimate
- Avoid recycling random number sequences
- Nine calls to Random() per 3×3 matrix $\rightarrow Nm/9 \cong 239000000$
- For initial seed 987654321 and $N = 200000000$,

$$Pr(\mathcal{A}) \cong 0.05017347$$

Point Estimate Considerations

- How many significant digits should be reported?
- Solution: run the simulation multiple times
- One option: use different initial seeds for each run
  - Caveat: Will the same squences of random numbers appear?
- Another option: use different a for each run
  - Caveat: Use a that gives a good random sequence
- For two runs with $a = 16807$ and 41214

$$Pr(\mathcal{A}) \cong 0.0502$$

Example 2: Craps

- Toss a pair of fair dice and sum the up faces
- If 7 or 11, win immediately
- If 2, 3, or 12, lose immediately
- Otherwise, sum becomes "point"
  - Roll until point is matched (win) or 7 (loss)
- What is $Pr(\mathcal{A})$, the probability of winning at craps?

Standard Craps Table

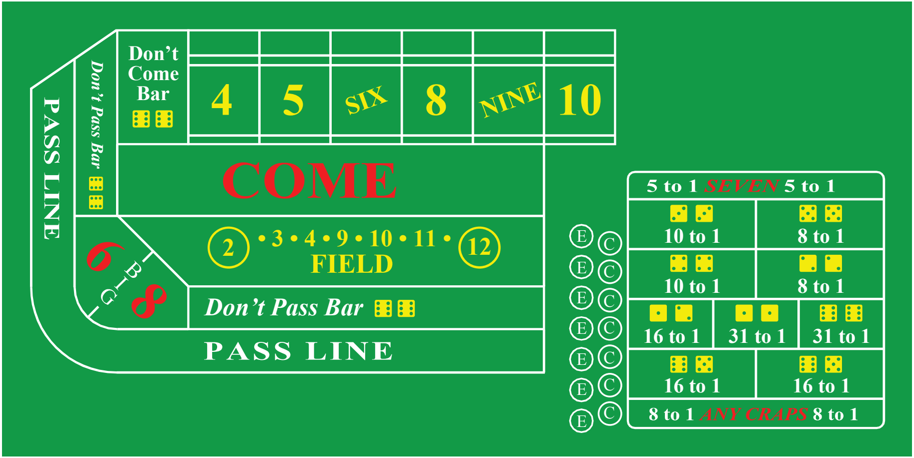


Figure retrieved from `http://en.wikipedia.org/wiki/File:Craps_table_layout.svg`

Craps: Axiomatic Solution

- Requires conditional probability
- Axiomatic solution: $244/495 \cong 0.493$
- Underlying mathematics must be changed if assumptions change
  - Example: unfair dice
- Axiomatic solution provides a nice consistency check for (easier) Monte Carlo simulation

Craps: Specification Model

- Model one die roll with Equilikely(1, 6)

Algorithm 2.4.1

```
wins = 0;
for (i = 1; i <= N; i++) {
    roll = Equilikely(1, 6) + Equilikely(1, 6);
    if (roll = 7 or roll = 11)
        wins++;
    else if (roll != 2 and roll != 3 and roll != 12) {
        point = roll;
        do {
            roll = Equilikely(1, 6) + Equilikely(1, 6);
            if (roll == point) wins++;
        } while (roll != point and roll != 7)
    }
} return (wins/N);
```

Craps: Computational Model

- Program craps: uses switch statement to determine rolls
- For $N = 10000$ and three different initial seeds (see text)

$$Pr(\mathcal{A}) = 0.497, 0.485, \text{and } 0.502$$

- These results are consistent with 0.493 axiomatic solution
- This (relatively) high probability is attractive to gamblers, yet ensures the house will win in the long run

Example 3: Hatcheck Girl

- Let $\mathcal{A}$ be that all checked hats are returned to wrong owners
- Without loss of generality, let the checked hats be numbered $1, 2, \ldots, n$
- The girl selects (equally likely) one of the remaining hats to return $\rightarrow n!$ permutations, each with probability $1/n!$
- Example: When $n = 3$ hats, possible return orders are

$$1, 2, 3 \qquad 1, 3, 2 \qquad 2, 1, 3 \qquad 2, 3, 1 \qquad 3, 1, 2 \qquad 3, 2, 1$$

- Only $2, 3, 1$ and $3, 1, 2$ correspond to all hats returned incorrectly

$$Pr(\mathcal{A}) = 1/3$$

Hatcheck: Specification Model

- Generate a random permutation of the first n integers
- The permutation corresponds to the order of hats returned

Clever Shuffling Algorithm (see Section 6.5)

```
for (i = 0; i < n - 1; i++) {
    j = Equilikely(i, n - 1);
    hold = a[j];
    a[j] = a[i]; /* swap a[i] and a[j] */
    a[i] = hold;
}
```

Generates a random permutation of an array a

- Check the permuted array to see if any element matches its index

Hatcheck: Computational Model

- Program hat: Monte Carlo simulation of hatcheck problem
- Uses shuffling algorithm to generate random permutation of hats
- For $n = 10$ hats, 10000 replications, and three different seeds

$$Pr(\mathcal{A}) = 0.369, 0.369, \text{and } 0.368$$

- What happens to the probability as $n \to \infty$?
- If using simulation, how big should n be?
  Instead, consider axiomatic solution

Hatcheck: Axiomatic Solution

- The probability $Pr(\mathcal{A})$ of no hat returned correctly is

$$1 - \left(1 - \frac{1}{2!} + \frac{1}{3!} - \ldots + (-1)^{n+1}\frac{1}{n!}\right)$$

- for $n = 10$, $Pr(\mathcal{A}) \cong 0.36787946$
- Important consistency check for validating craps
- As $n \to \infty$, the probability of no hat returned is

$$1/e \cong 0.36787944$$

In-Class Exercise L4-5

- Design an approach to show that the shuffle algorithm in slide 84 is correct.
- Implement the approach and graph the results.

Example 4: Stochastic Activity Network

- Activity durations are positive random variables
- n nodes, m arcs (activities) in the network
- Single source node (labeled 1), single terminal node (labeled n)
- Y_{ij} : positive random activity duration for arc a_{ij}
- T_j : completion time of all activities entering node j
- A path is critical with a certain probability

$$p(\pi_k) = Pr(\pi_k \equiv \pi_c), k = 1, 2, \ldots, r$$

Conceptual Model

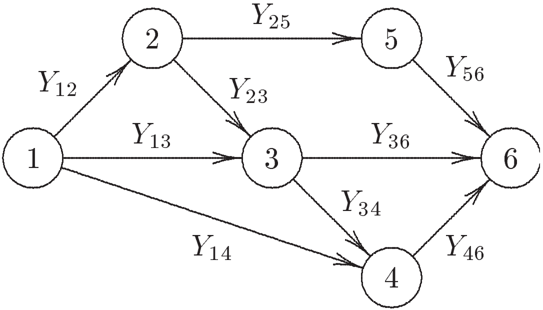
- Represent the network as an $n \times m$ node-arc incidence matrix N

$$N[i,j] = \begin{cases} 1 & \text{arc } j \text{leaves node } i \\ -1 & \text{arc } j \text{enters node } i \\ 0 & \text{otherwise} \end{cases}$$

- Use Monte Carlo simulation to estimate:
  - mean time to complete the network
  - probability that each path is critical

Conceptual Model

- Each activity duration is a uniform random variate



Example: Y_{12} has a *Uniform*$(0, 3)$ distribution

Specification Model

- Completion time T_j relates to incoming arcs

$$T_j = \max_{i \in \mathcal{B}(j)} \{T_i + Y_{ij}\} \quad j = 2, 3, \ldots, n$$

where $B(j)$ is the set of nodes immediately before node j

- Example: in the previous six-node example

$$T_6 = \max\{T_3 + Y_{36}, T_4 + Y_{46}, T_5 + Y_{56}\}$$

- We can write a recursive function to compute the T_j

Conceptual Model

- The previous 6-node, 9-arc network is represented as follows:

$$N = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & -1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 \end{bmatrix}$$

- In each row:
  - 1's represent arcs exiting that node
  - -1's represent arcs entering that node
- Exactly one 1 and one -1 in each column

Algorithm 2.4.2

- Returns a random time to complete all activities prior to node j for a single SAN with node-arc incidence matrix N

Algorithm 2.4.2

```
k = 1;
l = 0;
tmax = 0.0;
while (l < |$\mathcal{B}$(j)|) {
    if (N[j][k] == -1) {
        i = 1;
        while (N[j ][k] != 1)
            i++;
            t = Ti + Yi ;
            if (t >= $t_{max}$) $t_{max}$ = t;
            l++;
    }
    k++;
}
```

Computational Model

- Program *san*: MC simulation of a stochastic activity network
- Uses recursive function to compute completion times T_j (see text)
- Activity durations Y_{ij} are generated at random a priori
- Estimates T_n, the time to complete the entire network
- Computes critical path probabilities $p(\pi_k)$ for $k = 1, 2, \ldots, r$
- Axiomatic approach does not provide an analytic solution

Computational Model

- For 10000 realizations of the network and three initial seeds

$$T_6 = 14.64, 14.59, \text{and } 14.57$$

- Point estimates for critical path probabilities are

k	π_k	$\hat{p}_1(m_k)$	$\hat{p}_2(c_k)$	$\hat{p}_3(a_k)$	$\hat{p}_4(i_k)$
1	$\{a_{13}, a_{36}\}$	0.0168	0.0181	0.0193	0.0181
2	$\{a_{12}, a_{23}, a_{36}\}$	0.0962	0.0970	0.0904	0.0945
3	$\{a_{12}, a_{25}, a_{56}\}$	0.0013	0.0020	0.0013	0.0015
4	$\{a_{14}, a_{46}\}$	0.1952	0.1974	0.1907	0.1944
5	$\{a_{13}, a_{34}, a_{46}\}$	0.1161	0.1223	0.1182	0.1189
6	$\{a_{12}, a_{23}, a_{34}, a_{46}\}$	0.5744	0.5632	0.5801	0.5726

- Path π_6 is most likely to be critical – 57.26% of the time

Summary

- Random number generators
- Monte Carlo simulation and examples