Discrete-Event Simulation:

A First Course

Section 3.2: Multi-Stream Lehmer RNGs

Section 3.2: Multi-Stream Lehmer RNGs

- Typical DES models have many stochastic components
- Want a unique source of randomness for each component
- One (poor) option: multiple RNGs
- Better option: one RNG with multiple "streams" of random numbers

  one stream per stochastic component

- We will partition output from our Lehmer RNG into multiple streams

  0-13-142917-5

Example 3.2.1: ssq2 Arrival and Service Processes

- ssq2 has two stochastic components: arrival and service
- Allocate a different generator state variable to each

GetService with Unique Seed

```
double GetService(void)
{
    double s;
    static long x = 12345;
    PutSeed(x);
    s = Uniform(1.0, 2.0);
    GetSeed(&x);
    return (s);
}
```

- x represents the current state of the service process

Example 3.2.2: ssq2 Arrival and Service Processes

- Arrival should have its own static variable, initialized differently

GetArrival with Unique Seed

```
double GetArrival(void)
{
    static double arrival = START;
    static long x = 54321;
    PutSeed(x);
    arrival += Exponential(2.0);
    GetSeed(&x);
    return (arrival);
}
```

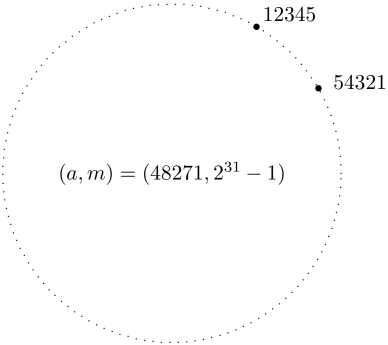
- `x` represents the current state of arrival process

The Modified Arrival and Service Processes

- As modified, arrival and service times are drawn from different streams of random numbers
- Nothing magic about the choice of seed for each stream
- The choices may, in fact, be poor ones!
- Provided the streams don't overlap, the processes are *uncoupled*
- Execution time cost is negligible (see Example 3.2.3)

Stream Considerations

- Potential problem: assignment of initial seeds to facilitate streams
- Each initial state should be chosen to produce *disjoint* streams
- If states are picked at whim, no guarantee of disjoint streams
- Some initial states may only be a few calls to `Random` apart



Jump Multipliers

- We will develop a multi-stream version of `rng`

Theorem (3.2.1)

Given $g(x) = ax \bmod m$ *and integer* $j (1 < j < m - 1)$

$$\textit{Jump function}: \quad g^j(x) = (a^j \bmod m)x \bmod m$$

$$\textit{Jump multiplier}: \quad a^j \bmod m$$

If $g(\cdot)$ *generates* $x_0, x_1, x_2, \ldots$ *then* $g^j(\cdot)$ *generates* $x_0, x_j, x_{2j}, \ldots$

- Theorem 3.2.1 is the key to creating streams

Example 3.2.4: An Example Jump Function

- If $m = 31$ and $a = 3$ and $j = 6$, the jump multiplier is

$$a^j \bmod m = 3^6 \bmod 31 = 16$$

- If $x_0 = 1$ then $g(x) = 3x \bmod 31$ generates

$$\underline{1},3,9,27,19,26,\underline{16},17,20,29,25,13,\underline{8},24,10,\ 30,28,22,\underline{4},\ldots$$

- The jump function $g^6(x) = 16x \bmod 31$ generates

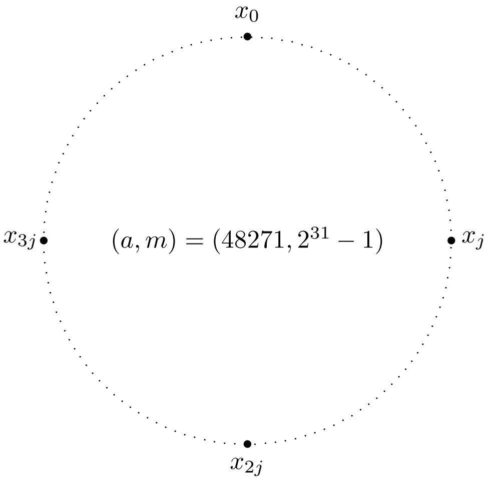
$$1,16,8,4,2,\ldots$$

- I.e., the first sequence is $x_0, x_1, x_2, \ldots$; the second is $x_0, x_6, x_{12}, \ldots$

Using the Jump Function

- First, compute the jump multiplier $a^j \bmod m$ (one time cost)
- Then, $g^j(\cdot)$ permits jumping from x_0 to x_j to x_{2j} to ...
- The user supplies *one* initial seed
- If j is chosen well, $g^j(\cdot)$ can “plant” additional initial seeds
- Each planted seed corresponds to a different stream
- Each planted seed is separated by j calls to `Random`

An Example 4-Stream Sequence



Example 3.2.5: An Appropriate Jump Multiplier

- Consider $256 = 2^8$ different streams of random numbers
- Partition the RNG output sequence into 256 disjoint subsequences of equal length
- Find the largest $j < 2^{31}/2^8 = 2^{23}$ such that the jump multiplier is modulus-compatible
- $g^j(x) = (48271^j \bmod m)x \bmod m$ can be implemented via Alg 2.2.1
- Then $g^j(x)$ can be used to plant the other 255 initial seeds
- Possibility of stream overlap is minimized (though not eliminated!)

Maximal Modulus-Compatible Jump Multipliers

- *Maximal jump multiplier*: $a^j \bmod m$ where j is the largest integer less than $\lfloor m/s \rfloor$ such that $a^j \bmod m$ is modulus compatible
- **Example 3.2.6**: multipliers for $(a, m) = (48271, 2^{31} - 1)$ RNG

# of streams s	$\lfloor m/s \rfloor$	jump size j	jump multiplier $a^j \bmod m$
1024	2097151	2082675	97070
512	4194303	4170283	44857
256	8388607	8367782	22925
128	16777215	16775552	40509

Library `rngs`

- `rngs` is an upward-compatible multi-stream replacement for `rng`
- By default, provides 256 streams, indexed 0 to 255 (0 is the default)
- Only one stream is active at any time
- Six available functions:
  - `Random(void)`
  - `PutSeed(long x)`: superseded by `PlantSeeds`
  - `GetSeed(long *x)`
  - `TestRandom(void)`
  - `SelectStream(int s)`: used to define the active stream
  - `PlantSeeds(long x)`: "plants" one seed per stream
- Henceforth, `rngs` is the library of choice

Example 3.2.7: ssq2 Revisited

- Use rngs functions for GetArrival, GetService

GetArrival Method

```
double GetArrival(void) {
    static double arrival = START;
    SelectStream(0);
    arrival += Exponential(2.0);
    return (arrival);
}
```

GetService Method

```
double GetService(void) {
    SelectStream(2);
    return (Uniform(1.0, 2.0));
}
```

- Include "rngs.h" and use PlantSeeds(12345)

Uncoupling Stochastic Processes

- Per modifications, arrival and service processes are uncoupled
- Consider changing the service process to

```
Uniform(0.0, 1.5) + Uniform(0.0, 1.5)
```

- Without uncoupling, arrival process sequence would change!
- With uncoupling, the service process “sees” *exactly* the same arrival sequence
- Important variance reduction technique

Single-Server Service Node with Multiple Job Types

- Extend the single-server service node model from Chapter 1
- Consider multiple job types, each with its own arrival and service process
- **Example 3.2.8**: Suppose there are two job types
  1. $\textit{Exponential}(4.0)$ interarrivals, $\textit{Uniform}(1.0, 3.0)$ service
  2. $\textit{Exponential}(6.0)$ interarrivals, $\textit{Uniform}(0.0, 4.0)$ service

  Use `rngs` to allocate a different stream to each stochastic process

Example 3.2.8: Arrival Process

Arrival Process

```
double GetArrival(int *j)
    /* Index j corresponds to job type */
{
    const double mean[2] = {4.0, 6.0};
    static double arrival[2] = {START, START};
    static int init = 1;
    double temp;
    if (init) {
        SelectStream(0);
        arrival[0] += Exponential(mean[0]);
        SelectStream(1);
        arrival[1] += Exponential(mean[1]);
        init = 0;
    }
    if (arrival[0] <= arrival[1])
        *j = 0;
    else
        *j = 1;
    temp = arrival[*j];
    SelectStream(*j);
    arrival[*j] += Exponential(mean[*j]);
    return (temp);
}
```

Example 3.2.8: Service Process

Service Process

```
double GetService(int j)
{
    const double min[2] = {1.0, 0.0};
    const double max[2] = {3.0, 4.0};
    SelectStream(j + 2);
    return (Uniform(min[j], max[j]));
}
```

- Index j matches service time to appropriate job type
- All four simulated stochastic processes are uncoupled
- Any process could be changed without altering the random sequence of others!

Consistency Checks

- With appropriate changes to `ssq2`, steady-state statistics are

$\bar{r}$	$\bar{w}$	$\bar{d}$	$\bar{s}$	$\bar{l}$	$\bar{q}$	$\bar{x}$
2.40	7.92	5.92	2.00	3.30	2.47	0.83

- Obvious consistency checks: $\bar{w} = \bar{d} + \bar{s}$ and $\bar{l} = \bar{q} + \bar{x}$
- Other consistency checks:
  - Both job types have avg service time of 2.0 $\Longrightarrow \bar{s} = 2.00$
  - Net arrival rate should be $1/4 + 1/6 = 5/12 \Longrightarrow \bar{r} = 12/5 = 2.40$
  - $\bar{x}$ should be ratio of arrival to service rates

$$\frac{5/12}{1/2} = 5/6 \cong 0.83$$