

3.2 Multi-Stream Lehmer Random Number Generators 111

A typical discrete-event simulation model will have many stochastic components. When this model is implemented at the computational level, the statistical analysis of system performance is often facilitated by having a unique source of randomness for each stochastic component. Although it may seem that the best way to meet this need for multiple sources of randomness is to create multiple random number generators, there is a simpler and better approach — use *one* random number generator to generate multiple “streams” of random numbers using multiple initial seeds as entry points, one for each stochastic system component. Consistent with this approach, in this section we extend the Lehmer random number generation algorithm from Chapter 2 by adding the ability to partition the generator’s output sequence into multiple subsequences (streams).

3.2.1 STREAMS

The library `rng` provides a way to partition the random number generator’s output into multiple streams by establishing multiple states for the generator, one for each stream. As illustrated by the following example, the function `PutSeed` can be used to set the state of the generator with the current state of the stream before generating a random variate appropriate to the corresponding stochastic component and the function `GetSeed` can be used to retrieve the revised state of the stream after the random variate has been generated.

Example 3.2.1 The program `ssq2` has two stochastic components, the arrival process and the service process, represented by the functions `GetArrival` and `GetService` respectively. To create a different stream of random numbers for each component, it is sufficient to allocate a different Lehmer generator state variable to each function. This is illustrated by modifying `GetService` from its original form in `ssq2`, which is

```
double GetService(void)                                /* original form */
{
    return (Uniform(1.0, 2.0));
}
```

to the multi-stream form indicated which uses the static variable `x` to represent the current state of the service process stream, initialized to 123456789.

```
double GetService(void)                                /* multi-stream form */
{
    double s;
    static long x = 123456789;                          /* use your favorite initial seed */
    PutSeed(x);                                          /* set the state of the generator */
    s = Uniform(1.0, 2.0);
    GetSeed(&x);                                         /* save the new generator state */
    return (s);
}
```

Example 3.2.2 As in the previous example, the function `GetArrival` should be modified similarly, with a corresponding static variable to represent the current state of the arrival process stream, but initialized to a *different* value. That is, the original form of `GetArrival` in program `ssq2`

```
double GetArrival(void)                                /* original form */
{
    static double arrival = START;
    arrival += Exponential(2.0);
    return (arrival);
}
```

should be modified to something like

```
double GetArrival(void)                                /* multi-stream form */
{
    static double arrival = START;
    static long x = 987654321;    /* use an appropriate initial seed */
    PutSeed(x);                  /* set the state of the generator */
    arrival += Exponential(2.0);
    GetSeed(&x);                  /* save the new generator state */
    return (arrival);
}
```

As in Example 3.2.1, in the multi-stream form the static variable `x` represents the current state of the arrival process stream, initialized in this case to 987654321. Note that there is nothing magic about this initial state (relative to 123456789) and, indeed, it may not even be a particularly good choice — more about that point later in this section.

If `GetService` and `GetArrival` are modified as in Examples 3.2.1 and 3.2.2, then the arrival times will be drawn from one stream of random numbers and the service times will be drawn from another stream. Provided the two streams don't overlap, in this way the arrival process and service process will be uncoupled.* As the following example illustrates, the cost of this uncoupling in terms of execution time is modest.

Example 3.2.3 The parameter `LAST` in program `ssq2` was changed to process 1 000 000 jobs and the execution time to process this many jobs was recorded. (A large number of jobs was used to get an accurate time comparison.) Program `ssq2` was then modified as in Examples 3.2.1 and 3.2.2 and used to process 1 000 000 jobs, with the execution time recorded. The increase in execution time was 20%.

* Also, because the scope of the two stream state variables (both are called `x`) is local to their corresponding functions, the use of `PutSeed` in program `ssq2` to initialize the generator can, and should, be eliminated from `main`.

Jump Multipliers

As illustrated in the previous examples, the library `rng` can be used to support the allocation of a unique stream of random numbers to each stochastic component in a discrete-event simulation program. There is, however, a potential problem with this approach — the assignment of initial seeds. That is, each stream requires a unique initial state that should be chosen to produce *disjoint* streams. But, if multiple initial states are picked at whim there is no convenient way to guarantee that the streams are disjoint; some of the initial states may be just a few calls to `Random` away from one another. With this limitation of the library `rng` in mind, we now turn to the issue of constructing a random number generation library called `rngs` which is a multi-stream version of the library `rng`. We begin by recalling two key points from Section 2.1.

- A Lehmer random number generator is defined by the function

$$g(x) = ax \bmod m,$$

where the modulus m is a large prime integer, the full-period multiplier a is modulus compatible with m , and $x \in \mathcal{X}_m = \{1, 2, \dots, m-1\}$.

- If x_0, x_1, x_2, \dots is an infinite sequence in \mathcal{X}_m generated by $g(x) = ax \bmod m$ then each x_i is related to x_0 by the equation

$$x_i = a^i x_0 \bmod m \quad i = 1, 2, \dots$$

The following theorem is the key to creating the library `rngs`. The proof is left as an exercise.

Theorem 3.2.1 Given a Lehmer random number generator defined by $g(x) = ax \bmod m$ and any integer j with $1 < j < m-1$, the associated *jump function* is

$$g^j(x) = (a^j \bmod m) x \bmod m$$

with the *jump multiplier* $a^j \bmod m$. For any $x_0 \in \mathcal{X}_m$, if the function $g(\cdot)$ generates the sequence x_0, x_1, x_2, \dots then the jump function $g^j(\cdot)$ generates the sequence x_0, x_j, x_{2j}, \dots

Example 3.2.4 If $m = 31$, $a = 3$, and $j = 6$ then the jump multiplier is

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

Starting with $x_0 = 1$ the function $g(x) = 3x \bmod 31$ generates the sequence

$$\underline{1}, 3, 9, 27, 19, 26, \underline{16}, 17, 20, 29, 25, 13, \underline{8}, 24, 10, 30, 28, 22, \underline{4}, 12, 5, 15, 14, 11, \underline{2}, 6, \dots$$

while the jump function $g^6(x) = 16x \bmod 31$ generates the sequence of underlined terms

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

That is, the first sequence is x_0, x_1, x_2, \dots and the second sequence is x_0, x_6, x_{12}, \dots

The previous example illustrates that once the jump multiplier $a^j \bmod m$ has been computed — this is a one-time cost — then the jump function $g^j(\cdot)$ provides a mechanism to jump from x_0 to x_j to x_{2j} , etc. If j is properly chosen then the jump function can be used in conjunction with a user supplied initial seed to “plant” additional initial seeds, each separated one from the next by j calls to `Random`. In this way disjoint streams can be automatically created with the initial state of each stream dictated by the choice of just *one* initial state.

Example 3.2.5 There are approximately 2^{31} possible values in the full period of our standard $(a, m) = (48271, 2^{31} - 1)$ Lehmer random number generator. Therefore, if we wish to maintain $256 = 2^8$ streams of random numbers (the choice of 256 is largely arbitrary) it is natural to partition the periodic sequence of possible values into 256 disjoint subsequences, each of equal length. This is accomplished by finding the largest value of j less than $2^{31}/2^8 = 2^{23} = 8388608$ such that the associated jump multiplier $48271^j \bmod m$ is modulus-compatible with m . Because this jump multiplier is modulus-compatible, the jump function

$$g^j(x) = (48271^j \bmod m) x \bmod m$$

can be implemented using Algorithm 2.2.1. This jump function can then be used in conjunction with one user supplied initial seed to efficiently plant the other 255 additional initial seeds, each separated one from the next by $j \cong 2^{23}$ steps.* By planting the additional seeds this way, the possibility of stream overlap is minimized.

Maximal Modulus-Compatible Jump Multipliers

Definition 3.2.1 Given a Lehmer random number generator with (prime) modulus m , full-period modulus-compatible multiplier a , and a requirement for s disjoint streams as widely separated as possible, the *maximal* jump multiplier is $a^j \bmod m$ where j is the largest integer less than $\lfloor m/s \rfloor$ such that $a^j \bmod m$ is modulus compatible with m .

Example 3.2.6 Consistent with Definition 3.2.1 and with $(a, m) = (48271, 2^{31} - 1)$ a table of maximal modulus-compatible jump multipliers can be constructed for 1024, 512, 256, and 128 streams, as illustrated.

# 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

Computation of the corresponding table for $a = 16807$ (the minimal standard multiplier) is left as an exercise.

* Because j is less than $2^{31}/2^8$, the last planted initial seed will be more than j steps from the first.

Library rngs

The library `rngs` is an upward-compatible multi-stream replacement for the library `rng`. The library `rngs` can be used as an alternative to `rng` in any of the programs presented earlier by replacing

```
#include "rng.h"
```

with

```
#include "rngs.h"
```

As configured `rngs` provides for 256 streams, indexed from 0 to 255, with 0 as the default stream. Although the library is designed so that all streams will be initialized to default values if necessary, the recommended way to initialize all streams is by using the function `PlantSeeds`. Only one stream is *active* at any time; the other 255 are *passive*. The function `SelectStream` is used to define the active stream. If the default stream is used exclusively, so that 0 is *always* the active stream, then the library `rngs` is functionally equivalent to the library `rng` in the sense that `rngs` will produce *exactly* the same Random output as `rng` (for the same initial seed, of course).

The library `rngs` provides six functions, the first four of which correspond to analogous functions in the library `rng`.

- `double Random(void)` — This is the Lehmer random number generator used throughout this book.
- `void PutSeed(long x)` — This function can be used to set the state of the active stream.
- `void GetSeed(long *x)` — This function can be used to get the state of the active stream.
- `void TestRandom(void)` — This function can be used to test for a correct implementation of the library.
- `void SelectStream(int s)` — This function can be used to define the active stream, i.e., the stream from which the next random number will come. The active stream will remain as the source of future random numbers until another active stream is selected by calling `SelectStream` with a different stream index `s`.
- `void PlantSeeds(long x)` — This function can be used to set the state of all the streams by “planting” a sequence of states (seeds), one per stream, with all states dictated by the state of the default stream. The following convention is used to set the state of the default stream:

if `x` is positive then `x` is the state;

if `x` is negative then the state is obtained from the system clock;

if `x` is 0 then the state is to be supplied interactively.

3.2.2 EXAMPLES

The following examples illustrate how to use the library `rngs` to allocate a separate stream of random numbers to each stochastic component of a discrete-event simulation model. We will see additional illustrations of how to use `rngs` in this and later chapters. From this point on `rngs` will be the basic random number generation library used for *all* the discrete-event simulation programs in this book.

Example 3.2.7 As a superior alternative to the multi-stream generator approach in Examples 3.2.1 and 3.2.2, the functions `GetArrival` and `GetService` in program `ssq2` can be modified to use the library `rngs`, as illustrated

```
double GetArrival(void)
{
    static double arrival = START;
    SelectStream(0);                /* this line is new */
    arrival += Exponential(2.0);
    return (arrival);
}

double GetService(void)
{
    SelectStream(2);                /* this line is new */
    return (Uniform(1.0, 2.0));
}
```

The other modification is to include `"rngs.h"` in place of `"rng.h"` and use the function `PlantSeeds(123456789)` in place of `PutSeed(123456789)` to initialize the streams.*

If program `ssq2` is modified consistent with Example 3.2.7, then the arrival process will be *uncoupled* from the service process. That is important because we may want to study what happens to system performance if, for example, the `return` in the function `GetService` is replaced with

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

Although two calls to `Random` are now required to generate each service time, this new service process “sees” *exactly* the same job arrival sequence as did the old service process. This kind of uncoupling provides a desirable variance reduction technique when discrete-event simulation is used to compare the performance of different systems.

* Note that there is nothing magic about the use of `rngs` stream 0 for the arrival process and stream 2 for the service process — any two different streams can be used. In particular, if even more separation between streams is required then, for example, streams 0 and 10 can be used.

A Single-Server Service Node With Multiple Job Types

A meaningful extension to the single-server service node model is *multiple job types*, each with its own arrival and service process. This model extension is easily accommodated at the conceptual level; each arriving job carries a job type that determines the kind of service provided when the job enters service. Similarly, provided the queue discipline is FIFO the model extension is straightforward at the specification level. Therefore, using program `ssq2` as a starting point, we can focus on the model extension at the implementation level. Moreover, we recognize that to facilitate the use of common random numbers, the library `rngs` can be used with a different stream allocated to each of the stochastic arrival and service processes in the model. The following example is an illustration.

Example 3.2.8 Suppose that there are two job types arriving independently, one with *Exponential*(4.0) interarrivals and *Uniform*(1.0, 3.0) service times and the other with *Exponential*(6.0) interarrivals and *Uniform*(0.0, 4.0) service times. In this case, the arrival process generator in program `ssq2` can be modified as

```
double GetArrival(int *j)                /* j denotes job type */
{
    const double mean[2]    = {4.0, 6.0};
    static double arrival[2] = {START, START};
    static int    init      = 1;
        double temp;
    if (init) {                          /* initialize the arrival array */
        SelectStream(0);
        arrival[0] += Exponential(mean[0]);
        SelectStream(1);
        arrival[1] += Exponential(mean[1]);
        init      = 0;
    }
    if (arrival[0] <= arrival[1])
        *j = 0;                          /* next arrival is job type 0 */
    else
        *j = 1;                          /* next arrival is job type 1 */
    temp = arrival[*j];                  /* next arrival time */
    SelectStream(*j);                   /* use stream j for job type j */
    arrival[*j] += Exponential(mean[*j]);
    return (temp);
}
```

Note that `GetArrival` returns the next arrival time *and* the job type as an index with value 0 or 1, as appropriate.

Example 3.2.9 As a continuation of Example 3.2.8, the corresponding service process generator in program `ssq2` can be modified as

```
double GetService(int j)
{
    const double min[2] = {1.0, 0.0};
    const double max[2] = {3.0, 4.0};
    SelectStream(j + 2);          /* use stream j + 2 for job type j */
    return (Uniform(min[j], max[j]));
}
```

Relative to Example 3.2.9, note that the job type index `j` is used in `GetService` to insure that the service time corresponds to the appropriate job type. Also, `rngs` streams 2 and 3 are allocated to job types 0 and 1 respectively. In this way all four simulated stochastic processes are uncoupled. Thus, the random variate model corresponding to any one of these four processes could be changed without altering the generated sequence of random variates corresponding to the other three processes.

Consistency Checks

Beyond the modifications in Examples 3.2.8 and 3.2.9, some job-type-specific statistics gathering needs to be added in `main` to complete the modification of program `ssq2` to accommodate multiple job types. If these modifications are made correctly, with *d.dd* precision the steady-state statistics that will be produced 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

The details are left in Exercise 3.2.4. How do we know these values are correct?

In addition to $\bar{w} = \bar{d} + \bar{s}$ and $\bar{l} = \bar{q} + \bar{x}$, the three following intuitive consistency checks give us increased confidence in these (estimated) steady-state results:

- Both job types have an average service time of 2.0, so that \bar{s} should be 2.00. The corresponding service rate is 0.5.
- The arrival rate of job types 0 and 1 are $1/4$ and $1/6$ respectively. Intuitively, the net arrival rate should then be $1/4 + 1/6 = 5/12$ which corresponds to $\bar{r} = 12/5 = 2.40$.
- The steady-state utilization should be the ratio of the arrival rate to the service rate, which is $(5/12)/(1/2) = 5/6 \cong 0.83$.

3.2.3 EXERCISES

Exercise 3.2.1 (a) Construct the $a = 16807$ version of the table in Example 3.2.6.
 (b) What is the $O(\cdot)$ time complexity of the algorithm you used?

Exercise 3.2.2^a (a) Prove that if m is prime, $1 \leq a \leq m - 1$, and $a^* = a^{m-2} \bmod m$ then

$$a^* a \bmod m = 1.$$

Now define

$$g(x) = ax \bmod m \quad \text{and} \quad g^*(x) = a^*x \bmod m$$

for all $x \in \mathcal{X}_m = \{1, 2, \dots, m - 1\}$. (b) Prove that the functions $g(\cdot)$ and $g^*(\cdot)$ generate the same sequence of states, except in *opposite* orders. (c) Comment on the implication of this relative to full period multipliers. (d) If $m = 2^{31} - 1$ and $a = 48271$ what is a^* ?

Exercise 3.2.3 Modify program `ssq2` as suggested in Example 3.2.7 to create two programs that differ only in the function `GetService`. For one of these programs, use the function as implemented in Example 3.2.7; for the other program, use

```
double GetService(void)
{
    SelectStream(2);                /* this line is new */
    return (Uniform(0.0, 1.5) + Uniform(0.0, 1.5));
}
```

(a) For both programs verify that *exactly* the same average interarrival time is produced (print the average with *d.dddddd* precision). Note that the average service time is approximately the same in both cases, as is the utilization, yet the service nodes statistics \bar{w} , \bar{d} , \bar{l} , and \bar{q} are different. (b) Why?

Exercise 3.2.4 Modify program `ssq2` as suggested in Examples 3.2.8 and 3.2.9. (a) What proportion of processed jobs are type 0? (b) What are \bar{w} , \bar{d} , \bar{s} , \bar{l} , \bar{q} , and \bar{x} for each job type? (c) What did you do to convince yourself that your results are valid? (d) Why are \bar{w} , \bar{d} , and \bar{s} the same for both job types but \bar{l} , \bar{q} , and \bar{x} are different?

Exercise 3.2.5 Prove Theorem 3.2.1.

Exercise 3.2.6 Same as Exercise 3.2.3, but using the `GetService` function in Example 3.1.4 instead of the `GetService` function in Exercise 3.2.3.

Exercise 3.2.7 Suppose there are three job types arriving independently to a single-server service node. The interarrival times and service times have the following characterization

job type	interarrival times	service times
0	<i>Exponential</i> (4.0)	<i>Uniform</i> (0.0, 2.0)
1	<i>Exponential</i> (6.0)	<i>Uniform</i> (1.0, 2.0)
2	<i>Exponential</i> (8.0)	<i>Uniform</i> (1.0, 5.0)

(a) What is the proportion of processed jobs for each type? (b) What are \bar{w} , \bar{d} , \bar{s} , \bar{l} , \bar{q} , and \bar{x} for each job type? (c) What did you do to convince yourself that your results are valid? (Simulate at least 100 000 processed jobs.)