1 Random Number Generation

The RNG class contains an implementation of the combined multiple recursive generator MRG32k3a proposed by L'Ecuyer [L'E99]. The C++ code was adapted from [LSCK01]. The MRG32k3a generator provides 1.8×10^{19} independent streams of random numbers, each of which consists of 2.3×10^{15} substreams. Each substream has a period (*i.e.*, the number of random numbers before overlap) of 7.6×10^{22}. The period of the entire generator is 3.1×10^{57}. Figure 1 provides a graphical idea of how the streams and substreams fit together.

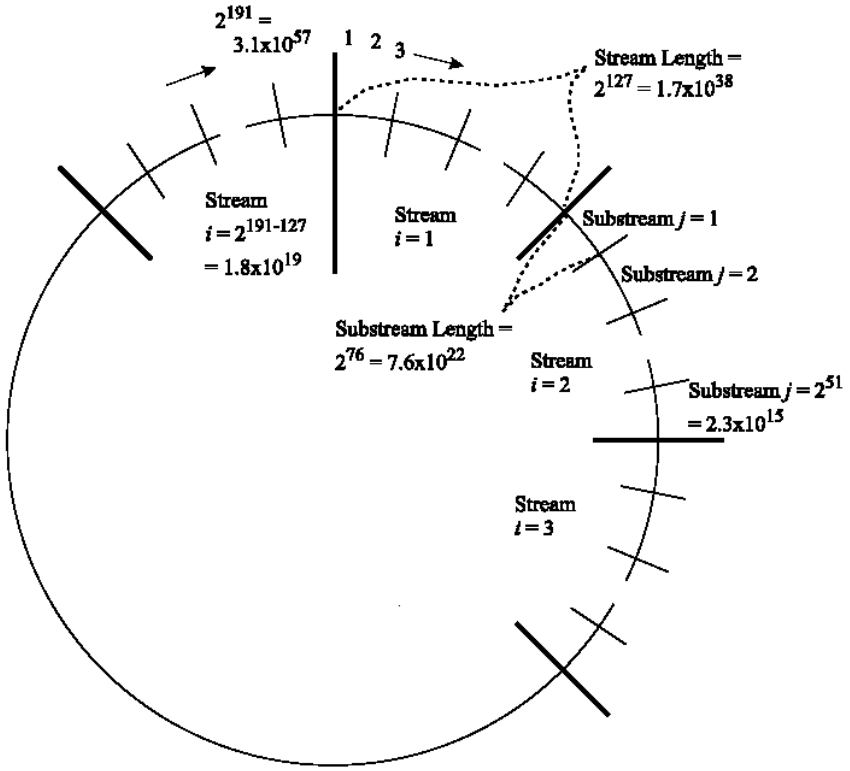

Figure 1: Overall arrangement of streams and substreams. [LSCK01]

A default RNG (`defaultRNG`), created at simulator initialization time, is provided. If multiple random variables are used in a simulation, each random variable should use a separate RNG object. When a new RNG object is created, it is automatically seeded to the beginning of the next independent stream of random numbers. Used in this manner, the implementation allows for a maximum of 1.8×10^{19} random variables.

Often, multiple independent replications of a simulation are needed (*i.e.*, to perform statistical analysis given multiple runs with fixed parameters). For each replication, a different substream should be used to ensure that the random number streams are independent. (This process is given as an OTcl example later.) This implementation allows

for a maximum of 2.3×10^{15} independent replications. Each random variable in a single replication can produce up to 7.6×10^{22} random numbers before overlapping.

Note: Only the most common functions are described here. For more information, see [LSCK01] and the source code found in `tools/rng.h` and `tools/rng.cc`. For a comparison of this RNG to the more common LCG16807 RNG (and why LCG16807 is not a good RNG), see [L'E01].

1.1 Seeding The RNG

Due to the nature of the RNG and its implementation, it is not necessary to set a seed (the default is 12345). If you wish to change the seed, functions are available. You should only set the seed of the default RNG. Any other RNGs you create are automatically seeded such that they produce independent streams. The range of valid seeds is 1 to `MAXINT`.

To get non-deterministic behavior, set the seed of the default RNG to 0. This will set the seed based on the current time of day and a counter. **This method should not be used to set seeds for independent replications.** There is no guarantee that the streams produced by two random seeds will not overlap. The only way to guarantee that two streams do not overlap is to use the substream capability provided by the RNG implementation.

1.1.1 Example

```
# Usage: ns rng-test.tcl [replication number]

if {$argc > 1} {
   puts "Usage: ns rng-test.tcl \[replication number\]"
   exit
}
set run 1
if {$argc == 1} {
   set run [lindex $argv 0]
}
if {$run < 1} {
   set run 1
}

# seed the default RNG
global defaultRNG
$defaultRNG seed 9999

# create the RNGs and set them to the correct substream
```

```
set arrivalRNG [new RNG]
set sizeRNG [new RNG]
for {set j 1} {$j < $run} {incr j} {
   $arrivalRNG next-substream
   $sizeRNG next-substream
}

# arrival_ is a exponential random variable describing the time
# between consecutive packet arrivals
set arrival_ [new RandomVariable/Exponential]
$arrival_ set avg_ 5
$arrival_ use-rng $arrivalRNG

# size_ is a uniform random variable describing packet sizes
set size_ [new RandomVariable/Uniform]
$size_ set min_ 100
$size_ set max_ 5000
$size_ use-rng $sizeRNG

# print the first 5 arrival times and sizes
for {set j 0} {$j < 5} {incr j} {
   puts [format "%-8.3f  %-4d" [$arrival_ value] \
            [expr round([$size_ value])]]
}
```

1.1.2 Output

```
% ns rng-test.tcl 1
6.358     4783
5.828     1732
1.469     2188
0.732     3076
4.002     626

% ns rng-test.tcl 5
0.691     1187
0.204     4924
8.849     857
2.111     4505
3.200     1143
```

1.2 OTcl Support

1.2.1 Commands

The following commands on the RNG class can be accessed from OTcl and are found in `tools/rng.cc`:

`seed` n – seed the RNG to n, if $n == 0$, the seed is set according to the current time and a counter

`next-random` – return the next random number

`seed` – return the current value of the seed

`next-substream` – advance to the next substream

`reset-start-substream` – reset the stream to the beginning of the current substream

`normal` avg std – return a number sampled from a normal distribution with the given average and standard deviation

`lognormal` avg std – return a number sampled from a lognormal distribution with the given average and standard deviation

The following commands on the RNG class can be accessed from OTcl and are found in `tcl/lib/ns-random.tcl`:

`exponential` mu – return a number sampled from an exponential distribution with mean mu

`uniform` min max – return an integer sampled from a uniform distribution on [min, max]

`integer` k – return an integer sampled from a uniform distribution on [0, k-1]

1.2.2 Example

```
# Usage: ns rng-test2.tcl [replication number]

if {$argc > 1} {
   puts "Usage: ns rng-test2.tcl \[replication number\]"
   exit
}
set run 1
if {$argc == 1} {
   set run [lindex $argv 0]
```

```
}
if {$run < 1} {
   set run 1
}

# the default RNG is seeded with 12345

# create the RNGs and set them to the correct substream
set arrivalRNG [new RNG]
set sizeRNG [new RNG]
for {set j 1} {$j < $run} {incr j} {
   $arrivalRNG next-substream
   $sizeRNG next-substream
}

# print the first 5 arrival times and sizes
for {set j 0} {$j < 5} {incr j} {
   puts [format "%-8.3f  %-4d" [$arrivalRNG lognormal 5 0.1] \
            [expr round([$sizeRNG normal 5000 100])]]
}
```

1.2.3 Output

```
% ns rng-test2.tcl 1
142.776   5038
174.365   5024
147.160   4984
169.693   4981
187.972   4982

% ns rng-test2.tcl 5
160.993   4907
119.895   4956
149.468   5131
137.678   4985
158.936   4871
```

1.3 C++ Support

1.3.1 Member Functions

The random number generator is implemented by the RNG class and is defined in `tools/rng.h`.

Note: The Random class in `tools/random.h` is an older interface to the standard random number stream.

Member functions provide the following operations:

`void set_seed (long seed)` – set the seed of the RNG, if *seed* == 0, the seed is set according to the current time and a counter

`long seed (void)` – return the current seed

`long next (void)` – return the next random number as an integer on [0, `MAXINT`]

`double next_double (void)` – return the next random number on [0, 1]

`void reset_start_substream (void)` – reset the stream to the beginning of the current substream

`void reset_next_substream (void)` – advance to the next substream

`int uniform (int k)` – return an integer sampled from a uniform distribution on [0, k-1]

`double uniform (double r)` – return a number sampled from a uniform distribution on [0, r]

`double uniform (double a, double b)` – return a number sampled from a uniform distribution on [a, b]

`double exponential (void)` – return a number sampled from an exponential distribution with mean 1.0

`double exponential (double k)` – return a number sampled from an exponential distribution with mean k

`double normal (double avg, double std)` – return a number sampled from a normal distribution with the given average and standard deviation

`double lognormal (double avg, double std)` – return a number sampled from a log-normal distribution with the given average and standard deviation

1.3.2 Example

```
/* create new RNGs */
RNG arrival (23456);
RNG size;

/* set the RNGs to the appropriate substream */
for (int i = 1; i < 3; i++) {
```

```
  arrival.reset_next_substream();
  size.reset_next_substream();
}

/* print the first 5 arrival times and sizes */
for (int j = 0; j < 5; j++) {
  printf ("%-8.3f  %-4d\n", arrival.lognormal(5, 0.1),
            int(size.normal(500, 10)));
}
```

1.3.3 Output

```
161.826   506
160.591   503
157.145   509
137.715   507
118.573   496
```

References

[L'E99] Pierre L'Ecuyer. Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

[L'E01] Pierre L'Ecuyer. Software for uniform random number generation: Distinguishing the good and the bad. In *Proceedings of the 2001 Winter Simulation Conference*, pages 95–105, December 2001.

[LSCK01] Pierre L'Ecuyer, Richard Simard, E. Jack Chen, and W. David Kelton. An object-oriented random number package with many long streams and substreams. *Operations Research*, 2001.