Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

SIMULATION IN JAVA WITH SSJ

Pierre L'Ecuyer
Eric Buist

Département d'Informatique et de Recherche Opérationnelle
Université de Montréal, C.P. 6128, Succ. Centre-Ville
Montréal, H3C 3J7, CANADA

ABSTRACT

We describe SSJ, an organized set of software tools offering general-purpose facilities for stochastic simulation programming in Java. It supports the event view, process view, continuous simulation, and arbitrary mixtures of these. Random number generators with multiple streams and substreams, quasi-Monte Carlo methods and their randomizations, and random variate generation for a rich selection of distributions, are all supported in an integrated framework. Performance, flexibility, and extensibility were key criteria in the design and implementation of SSJ. We illustrate its use by simple examples.

1 INTRODUCTION

In the early days of computing, simulation programs were written in a general-purpose language such as FORTRAN. Then came specialized languages like GPSS, Simscript, etc., devoted to discrete-event simulation. Nowadays, most commercial software products for simulation offer point-and-click graphical environments that permit one to specify a model and get the simulation program up and running without explicitly writing programming code. These environments are quite convenient from the user's viewpoint, because they do not require knowledge of a programming language, provide graphical animation, have automatic facilities to collect statistics and run experiments, and can sometimes perform some kind of optimization. On the other hand, these point-and-click tools are often too restrictive, because they are targeted at a limited class of models. With them, simulating a system whose logic is complicated or unconventional may become difficult. One must frequently revert to a general-purpose language to program the more complex aspects of a model or unsupported operations. Compilers and supporting tools for specialized languages are less widely available and cost more than for general purpose languages. The graphical and automatic devices also tend to slow down the simulation significantly. Fast execution times are important for example in a context of optimization, where thousands of variants of a base system may have to be simulated, or for on-line applications where a fast response time is needed. For example, when pricing financial derivatives by simulation, precise estimates are often required within a few seconds or minutes. To optimize the staffing and/or scheduling of a large telephone call center or other complex stochastic systems by simulation, a well-tuned efficient program may already run for several hours or even a few days, so slowing it down by a factor of 10 (say) makes a significant difference. The constant increase of cheap computing power will not change this: the complexity of models increases at least as fast as the speed of computers.

SSJ (which stands for *Stochastic Simulation in Java*) is an organized collection of Java packages whose purpose is to facilitate simulation programming in the general-purpose Java language. A early version was described by L'Ecuyer, Meliani, and Vaucher (2002). Advantages of programming in Java include: (1) greater flexibility than with graphical environments, (2) extensive and high-quality Java development tools, libraries, runtime optimizers, interfaces to other softwares, etc., (3) runs on practically any type of computer without change, and (4) programs run much faster than under typical point-and-click simulation environments.

The facilities offered are grouped into different packages, each one having its own user's guide, in the form of a PDF file, in addition to standard on-line documentation in HTML produced via `javadoc`. There is also a set of commented examples of simulation programs in a separate PDF document. An excellent way of learning more about SSJ is to study these examples. The tool is still under active development; new packages and methods are being added on a regular basis. It is developed and maintained at the Université de Montréal, and is available on-line from the first author's web page (L'Ecuyer 2004b).

SSJ is definitely not the only Java-based simulation framework. There are more than a dozen oth-

ers, including for instance *Silk* (Kilgore 2003), *J-Sim* (Tyan 2005), *JSIM* (Miller 2005), *Simkit* (Buss 2002), and *DSOL* (Jacobs and Verbraeck 2004). Our framework has a different design, in several aspects, from each of those.

In the next section we give an overview of SSJ. In Section 3, we illustrate some of the available facilities by a concrete example. Other examples can be found in L'Ecuyer, Meliani, and Vaucher (2002), Buist and L'Ecuyer (2005b), and in SSJ user's guide (L'Ecuyer 2004b).

2 OVERVIEW OF SSJ

We now describe the different packages that currently comprise SSJ. They provide probability distribution functions, goodness-of-fit tests for fitting distributions, uniform non-uniform random number generators, highly-uniform point sets that can replace the uniform random numbers, statistical collectors, event-list management tools for discrete-event simulation, and higher-level facilities for process-oriented simulation.

2.1 Package `probdist`

This package provides classes to handle probability distributions. Methods are available to compute mass, density, distribution, complementary distribution, and inverse distribution functions for discrete and continuous distributions. It does not directly generate random variates (package `randvar` does that) but its methods can be used together with a random number generator to generate random variates by inversion. It is useful not only for simulation, but for several other types of applications related to computational probability and statistics.

Standard distributions are implemented, each in its own class. Two types of methods are provided in most classes: *static methods*, for which no object needs to be created, and methods associated with *distribution objects*. Constructing an object from one of these classes can be convenient if the distribution function (or its inverse) has to be evaluated several times for the same distribution. In certain cases (for the Poisson distribution, for example), creating the distribution object precomputes tables that speed up significantly all subsequent method calls. This trades memory, plus a one-time setup cost, for speed. On the other hand, the static methods that do not require the creation of an object are sometimes more appropriate.

2.2 Package `gof`

The `gof` package contains tools for univariate goodness-of-fit (GOF) statistical tests, e.g., for testing the hypothesis $\mathcal{H}_0$ that a sample $X_1, \ldots, X_n$ comes from a given univariate probability distribution F. The available tests include the chi-square, Kolmogorov-Smirnov, Anderson-Darling, and Crámer-von Mises tests. Methods are available to apply various types of transformations to the observations, to compute the test statistics and their p-values for the continuous and discrete cases, and to format graphical plots and reports. These tools are adapted from the TestU01 package (L'Ecuyer and Simard 2002), used for statistical testing of random number generators.

2.3 Package `rng`

This package offers the basic facilities for generating uniform random numbers. It defines an interface called `RandomStream` and some implementations of that interface. The interface specifies that each `RandomStream` object provides a stream of random numbers partitioned into multiple substreams. Methods are available to jump between the substreams, as discussed in (L'Ecuyer and Côté 1991, L'Ecuyer, Simard, Chen, and Kelton 2002). Several implementations of this interface are available, each one using a specific backbone uniform random number generator (RNG) whose period length is typically partitioned into very long non-overlapping segments to provide the streams and substreams. These RNGs have period lengths ranging from (approximately) 2^{113} to 2^{19937}. They have different speeds for generating the numbers and for jumping ahead. Most are linear but some are nonlinear. A stream can generate uniform variates (real numbers) over the interval (0,1), uniform integers over a given range of values $\{i, \ldots, j\}$, and arrays of these. Our example in Section 3 will illustrate the usefulness of these streams and substreams. The same `RandomStream` interface is used as well for quasi-Monte Carlo point sets and sequences, in the package `hups`.

Other tools included in this package permit one to manage and synchronize several streams simultaneously and to apply automatic transformations to the output of a given stream (e.g., to get a stream that generates antithetic variates).

2.4 Package `hups`

This package implements *highly uniform point sets and sequences* (HUPS) over the s-dimensional unit hypercube $[0,1)^s$, and tools for their randomization. HUPS are also called *low-discrepancy point sets and sequences* are are used for *quasi-Monte Carlo* (QMC) and *randomized* QMC (RQMC) numerical integration (L'Ecuyer and Lemieux 2002, L'Ecuyer 2004a, Owen 1998, Niederreiter 1992).

A typical use of QMC or RQMC is to estimate an integral of the form

$$\mu = \int_{[0,1)^s} f(\mathbf{u}) d\mathbf{u} \tag{1}$$

for some integer s. Practically speaking, any mathematical expectation that can be estimated by simulation can be written in this way, usually for a very complicated f and sometimes for $s = \infty$. The vector $\mathbf{u} = (u_0, u_1, u_2, \ldots)$ represents the *stream* of i.i.d. $U(0,1)$ random numbers produced by the RNG underlying the simulation and s is an upper bound on the number of calls to the RNG. The *Monte Carlo* method estimates μ by

$$Q_n = \frac{1}{n}\sum_{i=0}^{n-1} f(\mathbf{u}_i), \tag{2}$$

which is the average of f over a set $P_n = \{\mathbf{u}_0, \ldots, \mathbf{u}_{n-1}\}$ of independent random points in $[0,1)^s$.

RQMC replaces the independent points $\mathbf{u}_i$ by a set of random points having the following properties: (a) each point $\mathbf{u}_i$ taken individually is uniformly distributed over $[0,1)^s$ and (b) the point set P_n is *more evenly distributed* over $[0,1)^s$ than independent random points. Condition (b) amounts to inducing negative dependence between the points and can be interpreted as a generalized antithetic variates approach (Wilson 1983, Ben-Ameur, L'Ecuyer, and Lemieux 2004). The aim is to reduce the variance of Q_n. Two important classes of methods for constructing such point sets are *digital nets* and *integration lattices* (Niederreiter 1992, Sloan and Joe 1994, L'Ecuyer and Lemieux 2000, L'Ecuyer and Lemieux 2002). Both are implemented in this package, in various flavors. Some are *infinite sequences* of points, of which the first n points can be extracted for different values of n. The set P_n can then be enlarged as needed by increasing n. Some point sets are also infinite-dimensional. Available constructions include the Hammersley point sets, Halton sequences, Sobol', Faure, and Niederreiter sequences, Niederreiter-Xing nets, Korobov lattice rules and arbitrary rank-1 lattice rules, recurrence-based digital nets, and digital nets constructed by coding theoretic techniques.

Several randomization methods that satisfy our requirements (a) and (b) are available for these point sets. Some of them, such as the random shift and digital random shift, apply to all point sets, whereas others such as affine matrix scrambling, striped matrix scrambling, starting a sequence at a random point, etc., apply to specific types of point sets (see, e.g., the user's guide and L'Ecuyer and Lemieux 2000, L'Ecuyer and Lemieux 2002, Owen 2003).

Certain types of transformations (deterministic and random) can be applied to point sets via predefined container classes that act as filters. One example of such a transformation is the baker's transform, which stretches each coordinate by a factor of two and folds the $[1,2)$ interval back over $(0,1]$ via the mapping $u \to 2-u$ (Hickernell 2002).

The base class for point sets is an abstract class named `PointSet`. Each point set can be viewed as a two-dimensional array whose element (i,j) contains $u_{i,j}$, the coordinate j of point i. In the implementations of typical point sets, the values $u_{i,j}$ are not stored explicitly in a two-dimensional array, but relevant information is organized so that the points and their coordinates can be generated efficiently.

To enumerate the successive points or the successive coordinates of a given point, we use *point set iterators*, which resemble the iterators defined in Java *collections*, except that they loop over bi-dimensional sets. These iterators must implement an interface named `PointSetIterator`. Each `PointSet` class has a method that returns an iterator of the correct type for this point set. Several independent iterators can coexist at any given time for the same point set. An important feature of the `PointSetIterator` interface is that it extends the `RandomStream` interface. This means that any point set iterator can be used in place of a random stream that is supposed to generate i.i.d. $U(0,1)$ random variables, anywhere in a simulation program. It then becomes very easy to replace the (pseudo)random numbers by the coordinates $u_{i,j}$ of a randomized HUPS without changing the internal code of the simulation program. The example in Section 3 illustrates this.

2.5 Package `randvar`

This package provides tools for non-uniform random variate generation, primarily from standard distributions. A generator can be obtained simply by pairing an arbitrary distribution object (from package `probdist`) with an arbitrary `RandomStream` object. The generic classes `RandomVariateGen` and `RandomVariateGenInt` do that for continuous and discrete distributions, respectively. For example, a `RandomVariateGen` object `g` can be constructed by providing a previously created beta distribution with specific parameters and a stream that generates the uniform random numbers. Then, each call to `g.nextDouble()` will return a beta random variate. By default, all random variates are generated by inversion. Note that the stream can very well be an iterator over an RQMC point set instead of a stream of pseudorandom numbers.

To generate random variates by other methods than inversion, specialized classes are provided for a variety of standard discrete and continuous distributions. For example, five different methods are provided to generate random variates from the standard normal distribution (inversion, Box-Muller, polar, Kindermann-Ramage, and acceptance-complement ratio). In many cases, the constructors of the specialized classes precompute constants and tables that depend on the specific parameter values of the distribution, to speed up the marginal cost of generating each random variate. *Static* methods in the specialized classes allow the

generation of random variates from specific distributions without constructing a `RandomVariateGen` object.

This package also provides an interface to the *UNURAN* (Universal Non-Uniform RANdom number generators) package, a rich library of C functions designed and implemented by Leydold and Hörmann (2002). This interface can be used to access distributions and generation methods not available directly in SSJ.

2.6 Package `stat`

This package provides basic tools for collecting statistics and computing confidence intervals. Objects of class `Tally` collect data that comes as a sequence of real-valued observations $X_1, X_2, \ldots$. It can compute sample averages, sample standard deviations and confidence intervals on the mean based on the normality assumption. The class `TallyStore` is similar, but it also stores the individual observations in an extensible array. The class `Accumulate` in package `simevents` computes integrals and averages with respect to time. This class is in package `simevents` because its operation depends on the simulation clock.

Statistical collectors are also available for arrays and matrices of observations. Some of them automatically compute the empirical covariances between observations. Tools to compute confidence intervals for functions of several means (e.g., a ratio of two means) via the *delta theorem* (Serfling 1980) are also available.

Most classes in package `stat` extend the `Observable` class of Java, which provides basic support for the *observer* design pattern (Gamma, Helm, Johnson, and Vlissides 1998) and facilitates the separation of data generation (by the simulation program) from data processing (for statistical reports and displays). This can be very helpful in particular in large simulation programs or libraries, where different objects may need to process the same data in different ways. An `Observable` object in Java maintains a list of registered `Observer` objects, and broadcasts information to all its registered observers whenever a new observation is given to the collector. Any object that implements the interface `Observer` can register as an observer.

2.7 Package `simevents`

Event scheduling for discrete-event simulations is managed by the "chief-executive" class `Sim`, contained in this package, which contains the simulation clock and the central monitor. Different implementations of the event list are offered to the user. The default implementation is a splay tree.

The class `Event` provides facilities for creating and scheduling events in the simulation. Each type of event must be defined by implementing an extension of this class. It must contain a method `actions()` which describes what happens when the event occurs. Events can be scheduled, rescheduled, cancelled, etc. Classes for collecting statistics that depend on the simulation clock (such as the time-average length of a queue, for example) are provided in this package. Another class provides elementary tools for continuous simulation, where certain variables vary continuously in time according to ordinary differential equations with respect to time.

2.8 Package `simprocs`

Process-oriented simulation is managed through this package. *Processes* can be seen as *active objects* whose behavior in time is described by their method `actions()`. In contrast with the corresponding `actions()` method of *events*, that of processes is generally not executed instantaneously in the simulation time frame. A process class must be defined as a subclass of the abstract class `SimProcess` and implement the `actions()` method. The processes can be created, started, and killed. They can interact, can be suspended and resumed, can be waiting for a given resource or a given condition, etc. These processes may represent "autonomous" objects such as machines and robots in a factory, customers in a retail store, vehicles in a transportation or delivery system, etc. The process-oriented paradigm is a natural way of describing complex systems (Franta 1977, Law and Kelton 2000) and often leads to more compact code than the event-oriented view. However, it is often preferred to use events only, because this gives a faster simulation program by avoiding the process-synchronization overhead (see the example at the end of this subsection). Most complex discrete-event systems are quite conveniently modeled only with events. In SSJ, events and processes can be mixed freely. The processes actually use events for their synchronization.

The classes `Resource`, `Bin`, and `Condition` provide additional mechanisms for process synchronization. A `Resource` corresponds to a facility with limited capacity and a waiting queue. A process can request an arbitrary number of units of a resource, may have to wait until enough units are available, can use the resource for a certain time, and eventually releases it. A `Bin` supports producer/consumer relationships between processes. It corresponds essentially to a pile of free tokens and a queue of processes waiting for the tokens. A *producer* adds tokens to the pile whereas a *consumer* (a process) can ask for tokens. When not enough tokens are available, the consumer is blocked and placed in the queue. The class `Condition` supports the concept of processes waiting for a certain boolean condition to be true before continuing their execution.

Two implementations of processes are available in SSJ. The first uses Java threads as described in Section 4 of L'Ecuyer, Meliani, and Vaucher (2002). The second

is taken from DSOL (Jacobs and Verbraeck 2004). Unfortunately, none of them is fully satisfactory. Java threads are designed for *real parallelism*, not for the kind of *simulated* parallelism required in process-oriented simulation. In the Java Development Kit (JDK) 1.3.1 and earlier, *green threads* supporting simulated parallelism were available and our original implementation of processes described in L'Ecuyer, Meliani, and Vaucher (2002) was based on them. But green threads are no longer supported in recent Java runtime environments. The threads are true threads from the operating system. This adds significant overhead and prevents the use of a very large number of processes in the simulation. This implementation of processes with threads can be used safely only with the JDK version 1.3.1 or earlier. The second implementation, provided to us by Peter Jacobs, stays away from threads. It uses a Java reflexion mechanism that interprets the code of processes at run time and transforms everything into events. The implementation details are completely transparent to the user. There is no need to change the Java simulation program in any way, except for the `import` statement at the beginning of the program that decides which subpackage of `simprocs` we want to use.

To demonstrate the loss of performance incurred when using processes instead of events, we report on the timing of a simulation of an $M/M/1$ queue using events and processes. Customers arrive according to a Poisson process with rate $\lambda = 1$ and have exponential service times with mean $1/\mu$. When a customer arrives while the server is busy, it is put in a FIFO queue with unlimited capacity. In an event-driven simulation, each arrival and departure correspond to an event, and the waiting queue is represented by a list. In a process-driven simulation, each customer is represented by a process. The programs we used are very similar to those given in L'Ecuyer, Meliani, and Vaucher (2002).

Table 1 gives the CPU times, in seconds, to simulate the system for 10^5 time units (approximately 10^5 arrivals), for different values of μ and with three different implementations: with events (Events), processes implemented by Java threads (Threads), and the DSOL implementation of processes (DSOL). The column marked Queue reports the average queue size, to give an idea of the number of simultaneous processes. For $\mu \leq 1$, the queue is unstable so the queue size increases steadily with the simulation length. This permits us to examine the effect of a large number of simultaneous processes on the performance, for process-driven simulation. These times were obtained with an AMD Athlon processor running at 2.088GHz, with Java Runtime Environment (JRE) 1.5 running under Linux.

We see that the process-driven simulations are much slower than their event-driven counterparts, roughly by a factor of 12 for Java threads and a factor of 700 for DSOL. A greater queue size requires more internal node objects to store queued customers in a linked list, adding work for the virtual machine. This explains why CPU times for events increase when the system is unstable. With thread-based processes, the CPU times increase with the number of simultaneous processes. When that number is too large, the program eventually crashes (this is why no CPU time is reported), not because of insufficient memory but due to a limitation on the number of native threads that can be provided by the operating system. With the DSOL solution, the only limitation on the number of threads is memory size. As long as the queue is stable ($\mu > 1$), the execution time increases slowly with the average queue size. When the queue is unstable, decreasing μ reduces the number of events (ends of service) that occur during a given time period and this reduces execution times. The cost of the interpretation dominates the cost of object creation.

Table 1: Performance Comparison for Event-Driven and Process-Driven Simulation: CPU Times (in seconds) to Simulate 10^5 Customers

μ	Queue	Events	Threads	DSOL
2.0	0.50	0.11	1.29	73.5
1.8	0.71	0.12	1.31	75.8
1.6	1.09	0.12	1.37	77.6
1.4	1.77	0.12	1.43	80.5
1.2	4.04	0.11	1.54	85.6
1.0	272	0.11	4.44	101.8
0.8	10037	0.36	—	82.8
0.6	20059	0.34	—	64.8

3 EXAMPLE: AN ASIAN OPTION

We provide an elementary example that illustrates how to generate random numbers, exploit the multi-streams facilities, compute distribution functions, and collect elementary statistics with SSJ. In this example, we price an Asian option by simulation, using randomized quasi-Monte Carlo to reduce the variance. We also show how to estimate a sensitivity via finite differences with common random numbers.

3.1 The Model

A *geometric Brownian motion* (GBM) $\{S(\zeta),\ \zeta \geq 0\}$ satisfies $S(\zeta) = S(0)\exp\left[(r - \sigma^2/2)\zeta + \sigma B(\zeta)\right]$ where r is the *risk-free appreciation rate*, σ is the *volatility parameter*, and B is a standard Brownian motion, i.e., a process whose increments over disjoint intervals are independent normal random variables, with mean 0 and variance δ over an interval of length δ (see, e.g., Glasserman 2004). The GBM process is a popular model for the evolution in time of the market price of financial assets. A discretely-monitored

Asian option on the arithmetic average of a given asset has discounted payoff

$$X = e^{-rT} \max[\bar{S} - K,\ 0] \tag{3}$$

where K is a constant called the *strike price* and

$$\bar{S} = \frac{1}{s} \sum_{j=1}^{s} S(\zeta_j), \tag{4}$$

for some fixed observation times $0 < \zeta_1 < \cdots < \zeta_s = T$. The value (or fair price) of the Asian option is $v = E[X]$ where the expectation is taken under a so-called risk-neutral measure (Glasserman 2004).

3.2 Pricing by Monte Carlo

This v can be estimated by simulation as follows. Generate s independent and identically distributed (i.i.d.) $N(0, 1)$ random variables $Z_1, \ldots, Z_s$ and put $B(\zeta_j) = B(\zeta_{j-1}) + \sqrt{\zeta_j - \zeta_{j-1}} Z_j$, for $j = 1, \ldots, s$, where $B(\zeta_0) = \zeta_0 = 0$. Then, $S(\zeta_j) = S(0) e^{(r-\sigma^2/2)\zeta_j + \sigma B(\zeta_j)}$ for $j = 1, \ldots, s$ and the payoff can be computed via (3). This can be replicated n times, independently, and the option value is estimated by the average discounted payoff.

The Java program of Figures 1 and 2 implements this. Due to space limitations, we do not provide the most general and reusable program; for example, in a good design, the option (payoff function) and the underlying stochastic process might be defined in separate classes, the statistical collectors might be external and passed as parameters to the methods, etc. We have also removed certain parts of the program, including the `import` statements and some uninteresting instructions in the constructor.

The `Asian` constructor precomputes the discount factor e^{-rT} and the constants $\sigma\sqrt{\zeta_j - \zeta_{j-1}}$ and $(r - \sigma^2/2)(\zeta_j - \zeta_{j-1})$, that depend on the process parameters and observation times. The method `generatePath` generates the values of $S(\zeta_j)$ at the observation times, whereas `getPayoff` returns the corresponding option payoff. The method `simulateRuns` performs n independent simulation runs using the given random number stream and puts the n observations of the net payoff in the statistical collector `statValue`. In the `main` method, we first specify the $s = 10$ observation times $\zeta_j = (110 + j)/365$ for $j = 1, \ldots, s$, and put them in the array `zeta` (of size $s + 1$) together with $\zeta_0 = 0$. We then construct an `Asian` object with parameters $r = \log 1.09$, $\sigma = 0.2$, $K = 100$, $S(0) = 100$, $s = 12$, and the observation times contained in array `zeta`. We then perform 10^6 (one million) simulation runs, and print the results. This took approximately 6.6 seconds to run on a 2.088GHz computer, with JRE 1.5 running under Linux, and gave (5.848, 5.870) as 95% confidence interval on v. We have $\mathrm{Var}[X] \approx 61.4$ for this standard MC method.

3.3 Common Random Numbers

We now illustrate how the streams of SSJ are convenient for comparing two different configurations of a given system with common random numbers (CRNs) (Law and Kelton 2000). Let X_1 denote the payoff X for a given value of $\sigma = \sigma_1$ and X_2 the payoff when $\sigma = \sigma_1 + \delta$, for some small $\delta > 0$, and suppose we want to estimate $E[X_2 - X_1]$. This is useful, e.g., for estimating the sensitivity of the option price with respect to the volatility parameter σ (Glasserman 2004). If X_1 and X_2 are simulated with independent random numbers (IRNs), we have $\mathrm{Var}[X_2 - X_1] = \mathrm{Var}[X_1] + \mathrm{Var}[X_2] \approx 2\mathrm{Var}[X_1]$. Simulating them with CRNs means using exactly the same uniforms at exactly the same place for both X_1 and X_2, to make $\mathrm{Cov}[X_1, X_2] > 0$.

The method `compareWithCRN` performs n pairs of simulation runs with CRNs, using one substream of the given stream for each pair of runs. For each pair, we first generate the sample path for the current process and store the payoff X_1. The stream is then reset to the start of its current substream so that it will generate exactly the same sequence of random numbers when we generate the sample path for the second process, `p2`. The difference $X_2 - X_1$ is given as an observation to the statistical collector `statDiff`. Then the stream is advanced to the start of its next substream, ready for the next pair of runs.

Here the two processes make exactly s calls to the RNG for each run. In general, however, when comparing similar systems with CRNs, the number of calls to the RNG may be "random" and differ across systems. Even in that case, using the SSJ substreams as illustrated here ensures that the RNG starts at the same place for both systems and that the sequences of random numbers that are used do not overlap. For complex systems, different `RandomStream` objects can be used for different parts of the system (e.g., in a queueing network, perhaps one stream for the interarrival times and one stream for the service times at each service station) to maintain synchonization. Then, all streams must be reset to their appropriate substreams between the runs.

For this particular example, CRNs could be implemented by saving in an array the standard normal random variates produced by `NormalDist.inverseF01 (stream.nextDouble())` in `generatePath` and reusing them for the second process. This would be more efficient because there would be no need to generate the uniforms and invert the normal distribution twice. But for more complicated system, the random variates are often generated from different distributions across systems and/or it is typically very inconvenient to store the random variates

```
public class Asian {
   double strike;    // Strike price.
   int s;            // Number of observation times.
   double discount;  // Discount factor exp(-r * zeta[t]).
   double[] muDelta; // Differences * (r - sigma^2/2).
   double[] sigmaSqrtDelta; // Square roots of differences * sigma.
   double[] logS;    // Log of the GBM process: logS[t] = log (S[t]).
   // The array zeta[0..s+1] must contain zeta[0]=0.0, plus the s observation times.
   public Asian (double r, double sigma, double strike,
                 double s0, int s, double[] zeta) {
      ...
   }
   // Generates the process S.
   public void generatePath (RandomStream stream) {
       for (int j = 0; j < s; j++)
          logS[j+1] = logS[j] + muDelta[j] + sigmaSqrtDelta[j]
                    * NormalDist.inverseF01 (stream.nextDouble());
   }
   // Computes and returns the discounted option payoff.
   public double getPayoff () {
       double average = 0.0;  // Average of the GBM process.
       for (int j = 1; j <= s; j++) average += Math.exp (logS[j]);
       average /= s;
       if (average > strike) return discount * (average - strike);
       else return 0.0;
   }
   // Performs n indep. runs using stream and collects statistics in statValue.
   public void simulateRuns (int n, RandomStream stream, Tally statValue) {
      statValue.init();
      for (int i=0; i<n; i++) {
         generatePath (stream);     statValue.add (getPayoff ());
         stream.resetNextSubstream();
      }
   }
   // Estimates difference in option value between other process p2 and current process,
   // from n pairs of runs using common random numbers and given stream.
   public void compareWithCRN (Asian p2, int n, RandomStream stream,
                               Tally statValue, Tally statDiff) {
      double payoff1;
      for (int i=0; i<n; i++) {
         generatePath (stream);      statValue.add (payoff1 = getPayoff());
         stream.resetStartSubstream();
         p2.generatePath (stream);   statDiff.add (p2.getPayoff() - payoff1);
         stream.resetNextSubstream();
      }
   }
   public static void main (String[] args) {
      int s = 10;
      double[] zeta = new double[s+1];   zeta[0] = 0.0;
      for (int j=1; j<=s; j++) zeta[j] = (120.0 - s + j) / 365.0;
      Asian process = new Asian (Math.log (1.09), 0.2, 100.0, 100.0, s, zeta);
      Tally statValue = new Tally ("Stats on value of Asian option");
      Tally statDiff  = new Tally ("Stats on difference, with CRNs");
      int n = 1000000;
      process.simulateRuns (n, new MRG32k3a(), statValue);
      System.out.println (statValue.reportAndConfidenceIntervalStudent (0.95));
      System.out.println ("Variance with MC: " + statValue.variance());
      System.out.println ("Total CPU time:      " + timer.format() + "\n");
      Asian process2 = new Asian (Math.log (1.09), 0.21, 100.0, 100.0, s, zeta);
      process.compareWithCRN (process2, n, new MRG32k3a(), statValue, statDiff);
      System.out.println (statDiff.reportAndConfidenceIntervalStudent (0.95));
      System.out.println ("Variance with IRN: " + statValue.variance() * 2.0);
      System.out.println ("Variance with CRN: " + statDiff.variance());
   }
}
```

Figure 1: Pricing an Asian Option on a GMB Process

```
public class AsianRQMC extends Asian {
  public AsianRQMC (double r, double sigma, double strike, double s0, int s, double[] zeta) {
      super (r, sigma, strike, s0, s, zeta);
  }
  public void simulateRQMC (int m, PointSet p, RandomStream noise,
                            Tally statValue, Tally statAver) {
     PointSetIterator stream = p.iterator ();
     statAver.init();
     for (int j=0; j<m; j++) {
        p.randomize (noise);
        stream.resetStartStream();
        simulateRuns (p.getNumPoints(), stream, statValue);
        statAver.add (statValue.average());
     }
  }
  // Compares MC and RQMC variances.
  public void experimentRQMC (int m, PointSet p, RandomStream noise, double varMC,
                              Tally statValue, Tally statAver) {
     Chrono timer = new Chrono();
     simulateRQMC (m, p, noise, statValue, statAver);
     System.out.println ("Average with QMC:  " + statAver.average());
     System.out.println ("Variance with MC:  " + varMC);
     System.out.println ("Variance with QMC: " + p.getNumPoints() * statAver.variance());
     System.out.println ("Total CPU time:      " + timer.format() + "\n");
  }
  public static void main (String[] args) {
     int s = 10;
     double[] zeta = new double[s+1];   zeta[0] = 0.0;
     for (int j=1; j<=s; j++) zeta[j] = (120.0 - s + j) / 365.0;
     AsianRQMC process = new AsianRQMC (Math.log (1.09), 0.2, 100.0, 100.0, s, zeta);
     RandomStream noise = new MRG32k3a();
     Tally statValue = new Tally ("Stats on value of Asian option");
     Tally statAver = new Tally ("Stats on averages over RQMC point sets");
     int n = 100000;  // Number of runs for MC.
     process.simulateRuns (n, noise, statValue);
     double varMC = statValue.variance();
     int m = 30;   // Number of independent randomizations for QMC.
     DigitalNetBase2 p1 = new SobolSequence (16, 31, s); // 2^{16} points.
     p1.leftMatrixScramble (noise);
     System.out.println ("RQMC with Sobol point set with " + p1.getNumPoints() +
         " points, with affine matrix scramble and random digital shift:\n");
     process.experimentRQMC (m, p1, noise, varMC, statValue, statAver);
     PointSet p2 = new BakerTransformedPointSet (new LCGPointSet (65521, 944));
     System.out.println ("RQMC with Korobov lattice with " + p2.getNumPoints() +
         " points, with random shift and baker transform:\n");
     process.experimentRQMC (m, p2, noise, varMC, statValue, statAver);
  }
}
```

Figure 2: Pricing the Asian Option with RQMC

across successive simulation runs. The approach we took here works generally and requires no explicit storage.

This part of the program ran in about 10.83 seconds and provided (0.212, 0.214) as a 95% confidence interval on $E[X_2 - X_1]$ with CRNs. The variance of $X_2 - X_1$ is 0.1436 with CRNs and 122.6 with IRNs. So CRNs improves the efficiency roughly by a factor of 854.

3.4 Randomized Quasi-Monte Carlo

The program in Figure 2 extends `Asian` to `AsianRQMC`, which estimates the option value via randomized quasi-Monte Carlo (RQMC), as in L'Ecuyer (2004a). The method `simulateRQMC` makes m independent randomizations of an RQMC point set `p` of cardinality n and dimension s, randomized using the stream `noise`. For each randomization, it performs n simulation runs, one for each point of `p`. For that, it uses an iterator over `p`, called `stream`, as the underlying random number stream from which are generated the normal random variates. Each call to `resetNextSubstream` goes to the first coordinate of the next point and `resetStartStream` goes back to the first point. By taking the average over the n (correlated) simulations for each randomization, we get m i.i.d. observations of an unbiased estimator of v, collected in `statAver`. The method `experimentRQMC` uses this collector to compare the MC and RQMC variances. For a fair comparison, the variance with RQMC is multiplied by n to get a variance per simulation run, because each of the m observations is based on n simulation runs.

In the `main` routine, we first construct the `AsianRQMC` object and the random stream `noise` used to randomize the RQMC point sets. The program then estimates the MC variance with $n = 10^5$ and tries two RQMC point sets with $m = 30$ independent randomizations: (1) a digital net in base 2 defined as the first $n = 2^{16}$ points of the Sobol' sequence in s dimensions to which we apply a left matrix scramble, plus a digital random shift at each randomization; (2) a Korobov lattice with $n = 65521$ points (implemented as a linear congruential generator with modulus 65521 and multiplier 944) randomized by a random shift modulo 1, and to which we apply a baker transformation after the randomization. Note that `p.randomize()` in `simulateRQMC` performs a digital shift for digital nets and a random shift modulo 1 for lattice rules.

Each RQMC experiment makes $30n$ (approximately 2 million) simulation runs and takes about 7.4 seconds to run, compared with 6.6 seconds for one million runs for MC. The variance per run is approximately 0.00629 for the Sobol' net, 0.00258 for the Korobov rule with baker's transform, and 61.3 for standard MC. Thus, the second RQMC method improves the efficiency by a factor of $(2 \times 6.6/7.4) \times 61.3/0.00258 \approx 42382$.

We urge the reader to replicate this experiment in her/his favorite simulation software environment to compare the simplicity of the code and the program speeds.

4 FUTURE AND CONCLUSION

SSJ provides a set of robust, efficient, and convenient basic tools for stochastic simulation. Programming with SSJ provides more flexibility than using a graphical point-and-click environment because all the power of a modern general-purpose programming environment is readily available, together with a rich variety of libraries and development tools, many of which come for free. The resulting programs also run significantly faster in general.

Among the strengths of SSJ compared with other simulation softwares, we can also mention the availability of quasi-Monte Carlo tools well integrated with standard uniform and non-uniform RNGs, uniform RNGs with multiple streams and substreams, a choice of efficient event-list implementations, and support for a large variety of probability distributions. Specialized packages for certain areas of applications are currently developed in our lab on top of the kernel described here. They include tools for simulation in finance and in telecommunications, a package for contact centers simulation (Buist and L'Ecuyer 2005a), and another for revenue management in the airline industry.

The SSJ system, together with its documentation and examples, is available from the first author's web page (L'Ecuyer 2004b).

ACKNOWLEDGMENTS

The following individuals have participated in the development of SSJ: Eric Buist, Chiheb Dkhil, Yves Edel, Regina H. S. Hong, Alexander Keller, Pierre L'Ecuyer, Étienne Marcotte, Lakhdar Meliani, Abdelazziz Milib, François Panneton, Richard Simard, Pierre-Alexandre Tremblay, Jean Vaucher. Its development has been supported by NSERC-Canada grant No. ODGP0110050, NATEQ-Québec grant No. 02ER3218, a Killam fellowship, and a Canada Research Chair to the first author.

REFERENCES

Ben-Ameur, H., P. L'Ecuyer, and C. Lemieux. 2004. Combination of general antithetic transformations and control variables. *Mathematics of Operations Research* 29 (4): 946–960.

Buist, E., and P. L'Ecuyer. 2005a. *ContactCenters: A Java library for simulating contact centers*. Software user's guide, forthcoming.

Buist, E., and P. L'Ecuyer. 2005b. A Java library for simulating contact centers. In *Proceedings of the 2005 Winter Simulation Conference*: IEEE Press. Forthcoming.

Buss, A. 2002. Component-based simulation modeling with Simkit. In *Proceedings of the 2002 Winter Simulation Conference*, ed. E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, 243–249: IEEE Press.

Franta, W. R. 1977. *The process view of simulation*. New York: North Holland.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1998. *Design patterns: Elements of reusable object-oriented software*. second ed. Reading, Mass.: Addison-Wesley.

Glasserman, P. 2004. *Monte Carlo methods in financial engineering*. New York: Springer-Verlag.

Hickernell, F. J. 2002. Obtaining $o(n^{-2+\epsilon})$ convergence for lattice quadrature rules. In *Monte Carlo and Quasi-Monte Carlo Methods 2000*, ed. K.-T. Fang, F. J. Hickernell, and H. Niederreiter, 274–289. Berlin: Springer-Verlag.

Jacobs, P. H. M., and A. Verbraeck. 2004. Single-threaded specification of process-interaction formalism in Java. In *Proceedings of the 2004 Winter Simulation Conference*, ed. R. G. Ingalls, M. D. Rosetti, J. S. Smith, and B. A. Peters, 1548–1555: IEEE Press.

Kilgore, R. A. 2003. Object-oriented simulation with SML and Silk in .NET and Java. In *Proceedings of the 2003 Winter Simulation Conference*, ed. S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, 218–224: IEEE Press.

Law, A. M., and W. D. Kelton. 2000. *Simulation modeling and analysis*. Third ed. New York: McGraw-Hill.

L'Ecuyer, P. 2004a. Quasi-Monte Carlo methods in finance. In *Proceedings of the 2004 Winter Simulation Conference*, ed. R. G. Ingalls, M. D. Rossetti, J. S. Smith, and B. A. Peters. Piscataway, New Jersey: IEEE Press.

L'Ecuyer, P. 2004b. *SSJ: A Java library for stochastic simulation*. Software user's guide, Available at <www.iro.umontreal.ca/~lecuyer>.

L'Ecuyer, P., and S. Côté. 1991. Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software* 17 (1): 98–111.

L'Ecuyer, P., and C. Lemieux. 2000. Variance reduction via lattice rules. *Management Science* 46 (9): 1214–1235.

L'Ecuyer, P., and C. Lemieux. 2002. Recent advances in randomized quasi-Monte Carlo methods. In *Modeling Uncertainty: An Examination of Stochastic Theory, Methods, and Applications*, ed. M. Dror, P. L'Ecuyer, and F. Szidarovszky, 419–474. Boston: Kluwer Academic Publishers.

L'Ecuyer, P., L. Meliani, and J. Vaucher. 2002. SSJ: A framework for stochastic simulation in Java. In *Proceedings of the 2002 Winter Simulation Conference*, ed. E. Yücesan, C.-H. Chen, J. L. Snowdon, and J. M. Charnes, 234–242: IEEE Press.

L'Ecuyer, P., and R. Simard. 2002. *TestU01: A software library in ANSI C for empirical testing of random number generators*. Software user's guide. Available at <www.iro.umontreal.ca/~lecuyer>.

L'Ecuyer, P., R. Simard, E. J. Chen, and W. D. Kelton. 2002. An object-oriented random-number package with many long streams and substreams. *Operations Research* 50 (6): 1073–1075.

Leydold, J., and W. Hörmann. 2002. *UNURAN—a library for universal non-uniform random number generators*. Available at <statistik.wu-wien.ac.at/unuran>.

Miller, J. A. 2005. JSIM: A java-based simulation and animation environment. Available from <chief.cs.uga.edu/~jam/jsim/>.

Niederreiter, H. 1992. *Random number generation and quasi-Monte Carlo methods*, Volume 63 of *SIAM CBMS-NSF Regional Conference Series in Applied Mathematics*. Philadelphia: SIAM.

Owen, A. B. 1998. Latin supercube sampling for very high-dimensional simulations. *ACM Transactions on Modeling and Computer Simulation* 8 (1): 71–102.

Owen, A. B. 2003. Variance with alternative scramblings of digital nets. *ACM Transactions on Modeling and Computer Simulation* 13 (4): 363–378.

Serfling, R. J. 1980. *Approximation theorems for mathematical statistics*. New York: Wiley.

Sloan, I. H., and S. Joe. 1994. *Lattice methods for multiple integration*. Oxford: Clarendon Press.

Tyan, H.-Y. 2005. J-Sim home page. Available on-line at <www.j-sim.org>.

Wilson, J. R. 1983. Antithetic sampling with multivariate inputs. *American Journal of Mathematical and Management Sciences* 3:121–144.

AUTHOR'S BIOGRAPHIES

PIERRE L'ECUYER is Professor in the Département d'Informatique et de Recherche Opérationnelle, at the Université de Montréal, Canada. He holds the Canada Research Chair in Stochastic Simulation and Optimization. His main research interests are random number generation, quasi-Monte Carlo methods, efficiency improvement via variance reduction, sensitivity analysis and optimization of discrete-event stochastic systems, and discrete-event simulation in general. He is an Area/Associate Editor for *ACM TOMACS*, *ACM TOMS*, and *Statistics and Computing*. He obtained the prestigious *E. W. R. Steacie* fellowship in 1995-97 and a *Killam* fellowship in 2001-03. His recent research articles are available on-line from his web page: <http://www.iro.umontreal.ca/~lecuyer>.

ERIC BUIST is a M.Sc. Student at the Université de Montréal. His main interests are software engineering, object-oriented programming, and simulation. His e-mail address is <buisteri@IRO.UMontreal.CA>.