SSJ User's Guide

Overview and Examples

Version: November 9, 2009

Pierre L'Ecuyer [1]

Canada Research Chair in Stochastic Simulation and Optimization
Département d'Informatique et de Recherche Opérationnelle
Université de Montréal, Canada

SSJ stands for *stochastic simulation in Java*. The SSJ library provides facilities for generating uniform and nonuniform random variates, computing different measures related to probability distributions, performing goodness-of-fit tests, applying quasi-Monte Carlo methods, collecting statistics, and programming discrete-event simulations with both events and processes. This document provides a very brief overview of this library and presents several examples of small simulation programs in Java, based on this library. The examples are commented in detail. They can serve as a good starting point for learning how to use SSJ. The first part of the guide gives very simple examples that do not need event or process scheduling. The second part contains examples of discrete-event simulation programs implemented with an *event view*, while the third part gives examples of implementations based on the *process view*.

[1]SSJ was designed and implemented in the Simulation laboratory of the Département d'Informatique et de Recherche Opérationnelle (DIRO), at the Université de Montréal, under the supervision of Pierre L'Ecuyer, with the contribution of Mathieu Bague, Sylvain Bonnet, Éric Buist, Yves Edel, Regina H. S. Hong, Alexander Keller, Pierre L'Ecuyer, Étienne Marcotte, Lakhdar Meliani, Abdelazziz Milib, François Panneton, Richard Simard, Pierre-Alexandre Tremblay, and 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 P. L'Ecuyer.

Contents

1 Introduction

The aim of this document is to provide an introduction to SSJ via a brief overview and a series of examples. The examples are collected in three groups:

(1) those that need no event or process scheduling;

(2) those based on the discrete-event simulation paradigm and implemented with an *event view* using the package `simevents`;

(3) those implemented with the *process view*, supported by the package `simprocs`.

Sections 3 to 5 of this guide correspond to these three groups. Some examples (e.g., the single-server queue) are carried across two or three sections to illustrate different ways of implementing the same model. The Java code of all these examples is available on-line from the SSJ web page (just type "SSJ iro" in Google).

While studying the examples, the reader can refer to the functional definitions (the APIs) of the SSJ classes and methods in the guides of the corresponding packages. Each package in SSJ has its own user's guide in the form of a `.pdf` document that contains the detailed API and complete documentation, and starts with an overview of one or two pages. We strongly recommend reading each of these overviews. We also recommend to refer to the `.pdf` versions of the guides, because they contain a more detailed and complete documentation than the `.html` versions, which are better suited for quick on-line referencing for those who are already familiar with SSJ.

2 Overview of SSJ

SSJ is an organized set of packages whose purpose is to facilitate stochastic simulation programming in the Java language. The facilities offered are grouped into different packages, each one having its own user's guide as a `.pdf` file. This is the official documentation. There is also a simplified on-line documentation in HTML format, produced via `javadoc`. Early descriptions of SSJ are given in [12, 11]. Some of the tools can also be used for modeling (e.g., selecting and fitting distributions). SSJ is still growing actively. New packages, classes, and methods will be added in forthcoming years and others will be refined.

The packages currently offered are the following:

`util` contains utility classes used in the implementation of SSJ, and which are often useful elsewhere. For example, there are timers (for CPU usage), utilities to read or format numbers and arrays from/to text, operations on binary vectors and matrices, some mathematical functions and constants, root-finding tools, facilities for SQL database interface, and so on.

`probdist` contains a set of Java classes providing methods to compute mass, density, distribution, complementary distribution, and inverse distribution functions for many discrete and continuous probability distributions, as well as estimating the parameters of these distributions.

`probdistmulti` contains a set of Java classes providing methods to compute mass, density, distribution, complementary distribution, for some multi-dimensionnal discrete and continuous probability distributions.

`rng` provides facilities for generating uniform random numbers over the interval $(0, 1)$, or over a given range of integer values, and other types of simple random objects such as random permutations.

`hups` provides classes implementing highly uniform point sets and sequences (HUPS), also called low-discrepancy sets and sequences, and tools for their randomization.

`randvar` provides a collection of classes for non-uniform random variate generation, primarily from standard distributions.

`randvarmulti` provides a collection of classes for random number generators for some multi-dimensional distributions.

`gof` contains tools for performing univariate goodness-of-fit (GOF) statistical tests.

`stat` provides elementary tools for collecting statistics and computing confidence intervals.

`stat.list` this subpackage of `stat` provides support to manage lists of statistical collectors.

`simevents` provides and manages the event-driven simulation facilities as well as the simulation clock. Can manage several simulations in parallel, in the same program.

`simevents.eventlist` this subpackage of `simevents` offers several kinds of event list implementations.

`simprocs` provides and manages the process-driven simulation facilities.

`functions` contains classes that allow one to pass an arbitrary function of one variable as argument to a method and to apply elementary mathematical operations on generic functions.

`functionfit` provides basic facilities for curve fitting and interpolation with polynomials.

`charts` provides tools for easy construction, visualization, and customization of *xy* plots, histograms, and empirical styled charts from a Java program.

`stochprocess` implements different kinds of stochastic processes.

Dependence on other libraries

SSJ uses some classes from other free Java libraries.

The Colt library, developed at the Centre Européen de Recherche Nucléaire (CERN) in Geneva [5], is a large library that provides a wide range of facilities for high performance scientific and technical computing in Java. SSJ uses the class `DoubleArrayList` from Colt in a few of its classes, namely in packages `stat` and `hups`. The reason is that this class provides a very efficient and convenient implementation of an (automatically) extensible array of `double`, together with several methods for computing statistics for the observations stored in the array (see, e.g., `Descriptive`). The Colt library is distributed with the SSJ package as **colt.jar**.

The **linear_algebra** library is based on public domain LINPACK routines. They were translated from Fortran to Java by Steve Verrill at the USDA Forest Products Laboratory Madison, Wisconsin, USA. This software is also in the public domain and is included in the SSJ distribution as the **Blas.jar** archive, which must be in the CLASSPATH environment variable. It is used only in the `probdist` package to compute maximum likelihood estimators.

The optimization package of Steve Verrill includes Java translations of the MINPACK routines [6] for nonlinear least squares problems as well as UNCMIN routines [14] for unconstrained optimization. They were translated from Fortran to Java by Steve Verrill and are in the public domain. They are included in the SSJ distribution as the **optimization.jar** archive, which must be in the CLASSPATH environment variable. It is used only in the `probdist` package to compute maximum likelihood estimators.

JFreeChart is a free Java library that can generate a wide variety of charts and plots for use in applications, applets and servlets. **JFreeChart** currently supports, amongst others, bar charts, pie charts, line charts, XY-plots, histograms, scatter plots and time series plots. It is distributed with SSJ as **jfreechart-*.jar**. JCommon is a free general purpose Java library containing many useful classes used by JFreeChart and other Java packages. It is distributed with SSJ as **jcommon-*.jar**. JFreeChart (and JCommon) are used in the

SSJ package **charts** to create different kinds of charts. Both of these jar's must be in the CLASSPATH environment variable.

SSJ also provides an interface to the UNURAN library for nonuniform random number generation [13], in the `randvar` package. UNURAN does not have to be installed to be used with SSJ, because it is linked statically with the appropriate SSJ native library. However, the UNURAN documentation will be required to take full advantage of the library.

3 Some Elementary Examples

We start with elementary examples that illustrate how to generate uniform and nonuniform random numbers, construct probability distributions, collect elementary statistics, and compute confidence intervals, compare similar systems, and use randomized quasi-Monte Carlo point sets, with SSJ.

The models considered here are quite simple and some of the performance measures can be computed by (more accurate) numerical methods rather than by simulation. The fact that we use these models to give a first tasting of SSJ should not be interpreted to mean that simulation is necessarily the best tool for them.

3.1 Collisions in a hashing system

We want to estimate the expected number of collisions in a hashing system. There are k locations (or addresses) and m distinct items. Each item is assigned a random location, independently of the other items. A *collision* occurs each time an item is assigned a location already occupied. Let C be the number of collisions. We want to estimate $\mathbb{E}[C]$, the expected number of collisions, by simulation. A theoretical result states that when $k \to \infty$ while $\lambda = m^2/(2k)$ is fixed, C converges in distribution to a Poisson random variable with mean λ. For finite values of k and m, we may want to approximate the distribution of C by the Poisson distribution with mean λ, and use Monte Carlo simulation to assess the quality of this approximation. To do that, we can generate n independent realizations of C, say $C_1, \ldots, C_n$, compute their empirical distribution and empirical mean, and compare with the Poisson distribution.

The Java program in Listing 1 simulates $C_1, \ldots, C_n$ and computes a 95% confidence interval on $\mathbb{E}[C]$. The results for $k = 10000$, $m = 500$, and $n = 100000$, are in Listing 2. The reported confidence interval is (12.25, 12.29), whereas $\lambda = 12.5$. This indicates that the asymptotic result underestimates $\mathbb{E}[C]$ by nearly 2%.

The Java program imports the SSJ packages `rng` and `stat`. It uses only two types of objects from SSJ: a `RandomStream` object, defined in the `rng` package, that generates a stream of independent random numbers from the uniform distribution, and a `Tally` object, from the `stat` package, to collect statistical observations and produce the report. In SSJ, `RandomStream` is actually just an interface that specifies all the methods that must be provided by its different implementations, which correspond to different brands of random streams (i.e., different types of uniform random number generators). The class `MRG32k3a`, whose constructor is invoked in the main program, is one such implementation of `RandomStream`. This is the one we use here. The class `Tally` provides the simplest type of statistical collector. It receives observations one by one, and after each new observation, it updates the number, average, variance, minimum, and maximum of the observations. At any time, it can return these statistics or compute a confidence interval for the theoretical mean of these observations, assuming that they are independent and identically distributed with the normal distribution. Other types of collectors that memorize the observations are also available in SSJ.

Listing 1: Simulating the number of collisions in a hashing system

```
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.stat.*;

public class Collision {
   int k;              // Number of locations.
   int m;              // Number of items.
   double lambda;      // Theoretical expectation of C (asymptotic).
   boolean[] used;     // Locations already used.

   public Collision (int k, int m) {
      this.k = k;
      this.m = m;
      lambda = (double) m * m / (2.0 * k);
      used = new boolean[k];
   }

   // Generates and returns the number of collisions.
   public int generateC (RandomStream stream) {
      int C = 0;
      for (int i = 0; i < k; i++) used[i] = false;
      for (int j = 0; j < m; j++) {
         int loc = stream.nextInt (0, k-1);
         if (used[loc]) C++;
         else used[loc] = true;
      }
      return C;
   }

   // Performs n indep. runs using stream and collects statistics in statC.
   public void simulateRuns (int n, RandomStream stream, Tally statC) {
      statC.init();
      for (int i=0; i<n; i++) statC.add (generateC (stream));
      statC.setConfidenceIntervalStudent();
      System.out.println (statC.report (0.95, 3));
      System.out.println (" Theoretical mean: " + lambda);
   }

   public static void main (String[] args) {
      Tally statC = new Tally ("Statistics on collisions");
      Collision col = new Collision (10000, 500);
      col.simulateRuns (100000, new MRG32k3a(), statC);
   }
}
```

The class `Collision` offers the facilities to simulate copies of C. Its constructor specifies k and m, computes λ, and constructs a boolean array of size k to memorize the locations used so far, in order to detect the collisions. The method `generateC` initializes the boolean array to `false`, generates the m locations, and computes C. The method `simulateRuns` first resets the statistical collector `statC`, then generates n independent copies of C and pass these n observations to the collector via the method `add`. The method `statC.report` computes a confidence interval from these n observations and returns a statistical report in the form of a character string. This report is printed, together with the value of λ.

Listing 2: Results of the program `Collision`

```
REPORT on Tally stat. collector ==> Statistics on collisions
    num. obs.     min          max       average    standard dev.
    100000       1.000      29.000       12.271       3.380
  95.0% confidence interval for mean (student): (     12.250,     12.292 )

 Theoretical mean: 12.5
```

3.2 Nonuniform variate generation and simple quantile estimates

The program of Listing 3 simulates the following artificial model. Define the random variable

$$X = Y_1 + \cdots + Y_N + W_1 + \ldots + W_M,$$

where N is Poisson with mean λ, M is geometric with parameter p, the Y_j's are gamma with parameters (α, β), the W_j's are lognormal with parameters (μ, σ), and all these random variables are independent. We want to generate n copies of X, say $X_1, \ldots, X_n$, and estimate the 0.10, 0.50, 0.90, and 0.99 quantiles of the distribution of X, simply from the quantiles of the empirical distribution.

The method `simulateRuns` generates n copies of X and pass them to a statistical collector of class `TallyStore`, that stores the individual observations. These observations are sorted in increasing order by invoking `quickSort`, and the appropriate empirical quantiles are printed, together with a short report.

Listing 3: Simulating nonuniform variates and observing quantiles

```
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.probdist.*;
import umontreal.iro.lecuyer.randvar.*;
import umontreal.iro.lecuyer.stat.*;

public class Nonuniform {
   // The parameter values are hardwired here to simplify the program.
   double lambda = 5.0;   double p = 0.2;
   double alpha = 2.0;    double beta = 1.0;
   double mu = 5.0;       double sigma = 1.0;

   RandomStream stream = new LFSR113();
   RandomVariateGenInt genN = new RandomVariateGenInt
               (stream, new PoissonDist (lambda));       // For N
   RandomVariateGen genY = new GammaAcceptanceRejectionGen
               (stream, new GammaDist (alpha, beta));    // For Y_j
   RandomVariateGen genW = new RandomVariateGen
               (stream, new LognormalDist (mu, sigma));  // For W_j

   // Generates and returns X.
   public double generateX () {
      int N;  int M;  int j;  double X = 0.0;
      N = genN.nextInt();
      M = GeometricDist.inverseF (p, stream.nextDouble());  // Uses static method
      for (j = 0; j < N; j++) X += genY.nextDouble();
      for (j = 0; j < M; j++) X += genW.nextDouble();
      return X;
   }

   // Performs n indep. runs and collects statistics in statX.
   public void simulateRuns (int n) {
      TallyStore statX = new TallyStore (n);
      for (int i=0; i<n; i++) statX.add (generateX ());
      System.out.println (statX.report ());
      statX.quickSort();
      double[] data = statX.getArray();
      System.out.printf ("0.10 quantile: %9.3f%n", data[(int)(0.10 * n)]);
      System.out.printf ("0.50 quantile: %9.3f%n", data[(int)(0.50 * n)]);
      System.out.printf ("0.90 quantile: %9.3f%n", data[(int)(0.90 * n)]);
      System.out.printf ("0.99 quantile: %9.3f%n", data[(int)(0.99 * n)]);
   }

   public static void main (String[] args) {
      (new Nonuniform ()).simulateRuns (10000);
   }
}
```

Listing 4: Results of the program `Nonuniform`

```
REPORT on Tally stat. collector ==> null
    num. obs.     min          max        average    standard dev.
     10000       0.000    13184.890     989.135     1261.431

0.10 quantile:     9.469
0.50 quantile:   553.019
0.90 quantile:  2555.540
0.99 quantile:  5836.938
```

To simplify the program, all the parameters are fixed as constants at the beginning of the class. This is simpler, but not recommended in general because it does not permit one to perform experiments with different parameter sets in a single program. Passing the parameters to the constructor as in Listing 1 would require more lines of code, but would provide more flexibility.

The class initialization also constructs a `RandomStream` of type `LFSR113` (this is a faster uniform generator that `MRG32k3a`), used to generate all the random numbers. For the generation of N, we construct a Poisson distribution with mean λ (without giving it a name), and pass it together with the random stream to the constructor of class `PoissonGen`. The returned object `genN` is random number generator that generate Poisson random variables with mean λ, via inversion. As similar procedure is used to construct `genY` and `genW`, which generate gamma and lognormal random variates, respectively. Note that a `RandomVariateGenInt` produces integer-valued random variates, while a `RandomVariateGen` produces real-valued random variates. For the gamma distribution, we use a special type of random number generator based on a rejection method, which is faster than inversion. These constructors precompute some (hidden) constants once for all, to speedup the random variate generation. For the Poisson distribution with mean λ, the constructor of `PoissonDist` actually precomputes the distribution function in a table, and uses this table to compute the inverse distribution function each time a Poisson random variate needs to be generated with this particular distribution. This is possible because all Poisson random variates have the same parameter λ. If a different λ was used for each variate, then we would use the static method of `PoissonDist` instead of constructing a Poisson distribution, otherwise we would have to reconstruct the distribution each time. The static method reconstructs part of the table each time, with the given λ, so it is slower if we want to generate several Poisson variates with the same λ. As an illustration, we use the static method to generate the geometric random variates (in `generateX`), instead of constructing a geometric distribution and variate generator. To generate M, we invoke the static method `inverseF` of the class `GeometricDist`, which evaluates the inverse geometric distribution function for a given parameter p and a given uniform random variate.

The results of this program, with $n = 10000$, are in Listing 4. We see that X has a coefficient of variation larger than 1, and the quantiles indicate that the distribution is skewed, with a long thick tail to the right. We have $X < 553$ about half the time, but values over several thousands are not uncommon. This probably happens when N or M takes a large value. There are also cases where $N = M = 0$, in which case $X = 0$.

3.3 A discrete-time inventory system

Consider a simple inventory system where the demands for a given product on successive days are independent Poisson random variables with mean λ. If X_j is the stock level at the beginning of day j and D_j is the demand on that day, then there are $\min(D_j, X_j)$ sales, $\max(0, D_j - X_j)$ lost sales, and the stock at the end of the day is $Y_j = \max(0, X_j - D_j)$. There is a revenue c for each sale and a cost h for each unsold item at the end of the day. The inventory is controlled using a (s, S) policy: If $Y_j < s$, order $S - Y_j$ items, otherwise do not order. When an order is made in the evening, with probability p it arrives during the night and can be used for the next day, and with probability $1-p$ it never arrives (in which case a new order will have to be made the next evening). When the order arrives, there is a fixed cost K plus a marginal cost of k per item. The stock at the beginning of the first day is $X_0 = S$.

We want to simulate this system for m days, for a given set of parameters and a given control policy (s, S), and replicate this simulation n times independently to estimate the expected profit per day over a time horizon of m days. Eventually, we might want to *optimize* the values of the decision parameters (s, S) via simulation, but we do not do that here. (In practice, this is usually done for more complicated models.)

Listing 5: A simulation program for the simple inventory system

```java
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;
import umontreal.iro.lecuyer.probdist.PoissonDist;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.*;

public class Inventory {

   double lambda;  // Mean demand size.
   double c;       // Sale price.
   double h;       // Inventory cost per item per day.
   double K;       // Fixed ordering cost.
   double k;       // Marginal ordering cost per item.
   double p;       // Probability that an order arrives.

   RandomVariateGenInt genDemand;
   RandomStream streamDemand = new MRG32k3a();
   RandomStream streamOrder  = new MRG32k3a();
   Tally statProfit          = new Tally ("stats on profit");

   public Inventory (double lambda, double c, double h,
                     double K, double k, double p) {
      this.lambda = lambda;
      this.c = c;  this.h = h;  this.K = K;  this.k = k;  this.p = p;
      genDemand = new PoissonGen (streamDemand, new PoissonDist (lambda));
   }

   // Simulates the system for m days, with the (s,S) policy,
   // and returns the average profit per day.
   public double simulateOneRun (int m, int s, int S) {
       int Xj = S, Yj;          // Stock in the morning and in the evening.
       double profit = 0.0;     // Cumulated profit.
       for (int j=0; j<m; j++) {
           Yj = Xj - genDemand.nextInt(); // Subtract demand for the day.
           if (Yj < 0) Yj = 0;            // Lost demand.
           profit += c * (Xj - Yj) - h * Yj;
```

```
         if ((Yj < s) && (streamOrder.nextDouble() < p)) {
            // We have a successful order.
            profit -= K + k * (S - Yj);
            Xj = S;
         }
         else
            Xj = Yj;
      }
      return profit / m;
   }

   // Performs n independent simulation runs of the system for m days,
   // with the (s,S) policy, and returns a report with a 90% confidence
   // interval on the expected average profit per day.
   public void simulateRuns (int n, int m, int s, int S) {
      for (int i=0; i<n; i++)
         statProfit.add (simulateOneRun (m, s, S));
   }

   public static void main (String[] args) {
      Chrono timer = new Chrono();
      Inventory system = new Inventory (100.0, 2.0, 0.1, 10.0, 1.0, 0.95);
      system.simulateRuns (500, 2000, 80, 200);
      system.statProfit.setConfidenceIntervalStudent();
      System.out.println (system.statProfit.report (0.9, 3));
      System.out.println ("Total CPU time: " + timer.format());
   }
}
```

Listing 5 gives a Java program, based on the SSJ library, that performs the required simulation for $n = 500$, $m = 2000$, $s = 80$, $S = 200$, $\lambda = 100$, $c = 2$, $h = 0.1$, $K = 10$, $k = 1$, and $p = 0.95$.

The `import` statements at the beginning of the program retrieve the SSJ packages/classes that are needed. The `Inventory` class has a constructor that initializes the model parameters (saving their values in class variables) and constructs the required random number generators and the statistical collector. To generate the demands D_j on successive days, we create (in the last line of the constructor) a random number stream and a Poisson distribution with mean λ, and then a Poisson random variate generator `genDemand` that uses this stream and this distribution. This mechanism will (automatically) precompute tables to ensure that the Poisson variate generation is efficient. This can be done because the value of λ does not change during the simulation. The random number stream `streamOrder`, used to decide which orders are received, and the statistical collector `statProfit`, are also created when the `Inventory` constructor is invoked. The code that invokes their constructors is outside the `Inventory` constructor, but it could have been inside as well. On the other hand, `genDemand` must be constructed inside the `Inventory` constructor, because the value of λ is not yet defined outside. The *random number streams* can be viewed as virtual random number generators that generate random numbers in the interval $[0, 1)$ according to the uniform probability distribution.

The method `simulateOneRun` simulates the system for m days, with a given policy, and returns the average profit per day. For each day, we generate the demand D_j, compute the stock Y_j at the end of the day, and add the sales revenues minus the leftover inventory costs to the profit. If $Y_j < s$, we generate a uniform random variable U over the interval $(0, 1)$ and an order of size $S - Y_j$ is received the next morning if $U < p$ (that is, with probability p). In case of a successful order, we pay for it and the stock level is reset to S.

The method `simulateRuns` performs n independent simulation runs of this system and returns a report that contains a 90% confidence interval for the expected profit. The main program constructs an `Inventory` object with the desired parameters, asks for n simulation runs, and prints the report. It also creates a timer that computes the total CPU time to execute the program, and prints it. The results are in Listing 6. The average profit per day is approximately 85. It took 0.39 seconds (on a 2.4 GHz computer running Linux) to simulate the system for 2000 days, compute the statistics, and print the results.

Listing 6: Results of the program `Inventory`

```
REPORT on Tally stat. collector ==> stats on profit
    num. obs.     min          max        average    standard dev.
       500      83.969      85.753      84.961       0.324
  90.0% confidence interval for mean (student): (    84.938,    84.985 )

Total CPU time: 0:0:0.39
```

In Listing 7, we extend the `Inventory` class to a class `InventoryCRN` that compares two sets of values of the inventory control policy parameters (s, S).

Listing 7: Comparing two inventory policies with common random numbers

```
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.Chrono;

public class InventoryCRN extends Inventory {

   Tally statDiff = new Tally ("stats on difference");

   public InventoryCRN (double lambda, double c, double h,
                        double K, double k, double p) {
      super (lambda, c, h, K, k, p);
   }

   public void simulateDiff (int n, int m, int s1, int S1, int s2, int S2) {
      statDiff.init();
      for (int i=0; i<n; i++) {
         double value1 = simulateOneRun (m, s1, S1);
         double value2 = simulateOneRun (m, s2, S2);
         statDiff.add (value2 - value1);
      }
   }

   public void simulateDiffCRN (int n, int m, int s1, int S1, int s2, int S2) {
      statDiff.init();
      streamDemand.resetStartStream();
      streamOrder.resetStartStream();
      for (int i=0; i<n; i++) {
         double value1 = simulateOneRun (m, s1, S1);
         streamDemand.resetStartSubstream();
         streamOrder.resetStartSubstream();
         double value2 = simulateOneRun (m, s2, S2);
         statDiff.add (value2 - value1);
         streamDemand.resetNextSubstream();
         streamOrder.resetNextSubstream();
      }
   }

   public static void main (String[] args) {
      InventoryCRN system = new InventoryCRN (100.0, 2.0, 0.1, 10.0, 1.0, 0.95);
      Chrono timer = new Chrono();

      system.simulateDiff (5000, 200, 80, 198, 80, 200);
      system.statDiff.setConfidenceIntervalStudent();
      System.out.println (system.statDiff.report (0.9, 3));
      double varianceIndep = system.statDiff.variance();
      System.out.println ("Total CPU time: " + timer.format() + "\n");

      timer.init();
      system.simulateDiffCRN (5000, 200, 80, 198, 80, 200);
      System.out.println (system.statDiff.report (0.9, 3));
      double varianceCRN = system.statDiff.variance();
      System.out.println ("Total CPU time: " + timer.format());
      System.out.printf ("Variance ratio:  %8.4g%n", varianceIndep/varianceCRN);
   }
}
```

The method `simulateDiff` simulates the system with policies (s_1, S_1) and (s_2, S_2) independently, computes the difference in profits, and repeats this n times. These n differences are tallied in statistical collector `statDiff`, to estimate the expected difference in average daily profits between the two policies.

The method `simulateDiffCRN` does the same, but using *common random numbers* across pairs of simulation runs. After running the simulation with policy (s_1, S_1), the two random number streams are reset to the start of their current substream, so that they produce exactly the same sequence of random numbers when the simulation is run with policy (s_2, S_2). Then the difference in profits is given to the statistical collector `statDiff` as before and the two streams are reset to a new substream for the next pair of simulations.

Why not use the same stream for both the demands and orders? In this example, we need one random number to generate the demand each day, and also one random number to know if the order arrives, but only on the days where we make an order. These days where we make an order are not necessarily the same for the two policies. So if we use a single stream for both the demands and orders, the random numbers will not necessarily be used for the same purpose across the two policies: a random number used to decide if the order arrives in one case may end up being used to generate a demand in the other case. This can greatly diminish the power of the common random numbers technology. Using two different streams as in Listing 7 ensures at least that the random numbers are used for the same purpose for the two policies. For more explanations and examples about common random numbers, see [8, 10].

The main program estimates the expected difference in average daily profits for policies $(s_1, S_1) = (80, 198)$ and $(s_2, S_2) = (80, 200)$, first with independent random numbers, then with common random numbers. The other parameters are the same as before. The results are in Listing 8. We see that use of common random numbers reduces the variance by a factor of approximately 19 in this case.

Listing 8: Results of the program `InventoryCRN`

```
REPORT on Tally stat. collector ==> stats on difference
    num. obs.     min          max        average    standard dev.
      5000      -4.961       5.737        0.266        1.530
  90.0% confidence interval for mean (student): (     0.230,     0.302 )

Total CPU time: 0:0:0.56

REPORT on Tally stat. collector ==> stats on difference
    num. obs.     min          max        average    standard dev.
      5000      -1.297       2.124        0.315        0.352
  90.0% confidence interval for mean (student): (     0.307,     0.324 )

Total CPU time: 0:0:0.44
Variance ratio:     18.85
```

3.4 A single-server queue with Lindley's recurrence

We consider here a *single-server queue*, where customers arrive randomly and are served one by one in their order of arrival, i.e., *first in, first out* (FIFO). We suppose that the times between successive arrivals are exponential random variables with mean $1/\lambda$, that the service times are exponential random variables with mean $1/\mu$, and that all these random variables are mutually independent. The customers arriving while the server is busy must join the queue. The system initially starts empty. We want to simulate the first m customers in the system and compute the mean waiting time per customer.

This simple model is well-known in queuing theory: It is called an $M/M/1$ queue. Simple formulas are available for this model to compute the average waiting time per customer, average queue length, etc., over an *infinite* time horizon [7]. For a finite number of customers or a finite time horizon, these expectations can also be computed by numerical methods, but here we just want to show how it can be simulated.

In a single-server queue, if W_i and S_i are the waiting time and service time of the ith customer, and A_i is the time between the arrivals of the ith and $(i+1)$th customers, we have $W_1 = 0$ and the W_i's follow the recurrence

$$W_{i+1} = \max(0,\ W_i + S_i - A_i), \tag{1}$$

known as *Lindley's equation* [7].

Listing 9: A simulation based on Lindley's recurrence

```
import umontreal.iro.lecuyer.stat.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.probdist.ExponentialDist;
import umontreal.iro.lecuyer.util.Chrono;

public class QueueLindley {

   RandomStream streamArr  = new MRG32k3a();
   RandomStream streamServ = new MRG32k3a();
   Tally averageWaits = new Tally ("Average waits");

   public double simulateOneRun (int numCust, double lambda, double mu) {
      double Wi = 0.0;
      double sumWi = 0.0;
      for (int i = 2; i <= numCust; i++) {
         Wi += ExponentialDist.inverseF (mu, streamServ.nextDouble()) -
               ExponentialDist.inverseF (lambda, streamArr.nextDouble());
         if (Wi < 0.0) Wi = 0.0;
         sumWi += Wi;
      }
      return sumWi / numCust;
   }

   public void simulateRuns (int n, int numCust, double lambda, double mu) {
      averageWaits.init();
      for (int i=0; i<n; i++)
          averageWaits.add (simulateOneRun (numCust, lambda, mu));
   }
   public static void main (String[] args) {
      Chrono timer = new Chrono();
      QueueLindley queue = new QueueLindley();
      queue.simulateRuns (100, 10000, 1.0, 2.0);
      System.out.println (queue.averageWaits.report());
      System.out.println ("Total CPU time: " + timer.format());
   }
}
```

The program of Listing 9 exploits (1) to compute the average waiting time of the first m customers in the queue, repeats it n times independently, and prints a summary of the results. Here, for a change, we pass the model parameters to the methods instead of to the constructor, and the random variates are generated by static methods instead of via a `RandomVariateGen` object as in the *Inventory* class (previous example). This illustrates various ways of doing the same thing. The instruction "`Wi += ...`" could also be replaced by

```
    Wi += - Math.log (1.0 - streamServ.nextDouble()) / mu
          + Math.log (1.0 - streamArr.nextDouble()) / lambda;
```

which directly implements inversion of the exponential distribution.

3.5 Using the observer design pattern

Listing 10 adds a few ingredients to the program `QueueLindley`, in order to illustrate the *observer* design pattern implemented in package `stat`. This mechanism permits one to separate data generation from data processing. It can be very helpful in large simulation programs or libraries, where different objects may need to process the same data in different ways. These objects may have the task of storing observations or displaying statistics in different formats, for example, and they are not necessarily fixed in advance.

The *observer* pattern, supported by the `ObservationListener` interface in SSJ, offers the appropriate flexibility for that kind of situation. A statistical probe maintains a list of registered `ObservationListener` objects, and broadcasts information to all its registered observers whenever appropriate. Any object that implements the interface `ObservationListener` can register as an observer.

`StatProbe` in package `stat`, as well as its subclasses `Tally` and `Accumulate`, contains a list of `ObservationListener`'s. Whenever they receive a new statistical observation, e.g., via `Tally.add` or `Accumulate.update`, they send the new value to all registered observers. To register as an observer, an object must implement the interface `ObservationListener` This implies that it must provide an implementation of the method `newObservation`, whose purpose is to recover the information that the object has registered for.

In the example, the statistical collector `waitingTimes` transmits to all its registered listeners each new statistical observation that it receives via its `add` method. More specifically, each call to `waitingTimes.add(x)` generates in the background a call to `o.newObservation(waitingTimes, x)` for all registered observers `o`.

Two observers register to receive observations from `waitingTimes` in the example. They are anonymous objects of classes `ObservationTrace` and `LargeWaitsCollector`, respectively. Each one is informed of any new observation W_i via its `newObservation` method. The task of the `ObservationTrace` observer is to print the waiting times W_5, W_{10}, W_{15}, ..., whereas the `LargeWaitsCollector` observer stores in an array all waiting times that exceed 2. The statistical collector `waitingTimes` itself also stores appropriate information to be able to provide a statistical report when required.

The `ObservationListener` interface specifies that `newObservation` must have two formal parameters, of classes `StatProbe` and `double`, respectively. The second parameter is the value of the observation. In the case where the observer registers to several `ObservationListener` objects, the first parameter of `newObservation` tells it which one is sending the information, so it can adopt the correct behavior for this sender.

Listing 10: A simulation of Lindley's recurrence using observers

```
import java.util.*;
import umontreal.iro.lecuyer.stat.*;
import umontreal.iro.lecuyer.simevents.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;

public class QueueObs {

   Tally waitingTimes = new Tally ("Waiting times");
   Tally averageWaits = new Tally ("Average wait");
   RandomVariateGen genArr;
   RandomVariateGen genServ;
   int cust;    // Number of the current customer.

   public QueueObs (double lambda, double mu, int step) {
      genArr = new ExponentialGen (new MRG32k3a(), lambda);
      genServ = new ExponentialGen (new MRG32k3a(), mu);
      waitingTimes.setBroadcasting (true);
      waitingTimes.addObservationListener (new ObservationTrace (step));
      waitingTimes.addObservationListener (new LargeWaitsCollector (2.0));
   }

   public double simulateOneRun (int numCust) {
      waitingTimes.init();
      double Wi = 0.0;
      waitingTimes.add (Wi);
      for (cust = 2; cust <= numCust; cust++) {
         Wi += genServ.nextDouble() - genArr.nextDouble();
         if (Wi < 0.0) Wi = 0.0;
         waitingTimes.add (Wi);
      }
      return waitingTimes.average();
   }

   public void simulateRuns (int n, int numCust) {
      averageWaits.init();
      for (int i=0; i<n; i++)
          averageWaits.add (simulateOneRun (numCust));
   }

   public class ObservationTrace implements ObservationListener {
      private int step;

      public ObservationTrace (int step) { this.step = step; }

      public void newObservation (StatProbe probe, double x) {
         if (cust % step == 0)
            System.out.println ("Customer " + cust + " waited "
                     + x + " time units.");
      }
   }

   public class LargeWaitsCollector implements ObservationListener {
      double threshold;
      ArrayList<Double> largeWaits = new ArrayList<Double>();

      public LargeWaitsCollector (double threshold) {
         this.threshold = threshold;
      }

      public void newObservation (StatProbe probe, double x) {
         if (x > threshold) largeWaits.add (x);
      }

      public String formatLargeWaits () {
          // Should print the list largeWaits.
          return "not yet implemented...";
      }
   }

   public static void main (String[] args) {
      QueueObs queue = new QueueObs (1.0, 2.0, 5);
      queue.simulateRuns (2, 100);
      System.out.println ("\n\n" + queue.averageWaits.report());
   }
}
```

3.6 Pricing an Asian option

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 stochastic 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., [4]). 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

$$X = e^{-rT}\max[\bar{S} - K,\ 0] \tag{2}$$

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

$$\bar{S} = \frac{1}{t}\sum_{j=1}^{t} S(\zeta_j), \tag{3}$$

for some fixed observation times $0 < \zeta_1 < \cdots < \zeta_t = T$. The value (or fair price) of the Asian option is $c = E[X]$ where the expectation is taken under the so-called risk-neutral measure (which means that the parameters r and σ have to be selected in a particular way; see [4]).

This value c can be estimated by simulation as follows. Generate t independent and identically distributed (i.i.d.) $N(0,1)$ random variables $Z_1, \ldots, Z_t$ and put $B(\zeta_j) = B(\zeta_{j-1}) + \sqrt{\zeta_j - \zeta_{j-1}}Z_j$, for $j = 1, \ldots, t$, where $B(\zeta_0) = \zeta_0 = 0$. Then,

$$S(\zeta_j) = S(0)e^{(r-\sigma^2/2)\zeta_j + \sigma B(\zeta_j)} = S(\zeta_{j-1})e^{(r-\sigma^2/2)(\zeta_j - \zeta_{j-1}) + \sigma\sqrt{\zeta_j - \zeta_{j-1}}Z_j} \tag{4}$$

for $j = 1, \ldots, t$ and the payoff can be computed via (2). This can be replicated n times, independently, and the option value is estimated by the average discounted payoff. The Java program of Listing 11 implement this procedure.

Note that generating the sample path and computing the payoff is done in two different methods. This way, other methods could eventually be added to compute payoffs that are defined differently (e.g., based on the geometric average, or with barriers, etc.) over the same generated sample path.

Listing 11: Pricing an Asian option on a GMB process

```
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.probdist.NormalDist;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.*;

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]).

// 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) {
   this.strike = strike;
   this.s = s;
   discount = Math.exp (-r * zeta[s]);
   double mu = r - 0.5 * sigma * sigma;
   muDelta = new double[s];
   sigmaSqrtDelta = new double[s];
   logS = new double[s+1];
   double delta;
   for (int j = 0; j < s; j++) {
      delta = zeta[j+1] - zeta[j];
      muDelta[j] = mu * delta;
      sigmaSqrtDelta[j] = sigma * Math.sqrt (delta);
   }
   logS[0] = Math.log (s0);
}

// 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();
   }
}
```

```
   public static void main (String[] args) {
      int s = 12;
      double[] zeta = new double[s+1];   zeta[0] = 0.0;
      for (int j=1; j<=s; j++)
         zeta[j] = (double)j / (double)s;
      Asian process = new Asian (0.05, 0.5, 100.0, 100.0, s, zeta);
      Tally statValue = new Tally ("Stats on value of Asian option");

      Chrono timer = new Chrono();
      int n = 100000;
      process.simulateRuns (n, new MRG32k3a(), statValue);
      statValue.setConfidenceIntervalStudent();
      System.out.println (statValue.report (0.95, 3));
      System.out.println ("Total CPU time:      " + timer.format() + "\n");
   }
}
```

The method **simulateRuns** performs n independent simulation runs using the given random number stream and put the n observations of the net payoff in the statistical collector **statValue**. In the **main** program, we first specify the 12 observation times $\zeta_j = j/12$ for $j = 1, \dots, 12$, and put them in the array **zeta** (of size 13) together with $\zeta_0 = 0$. We then construct an **Asian** object with parameters $r = 0.05$, $\sigma = 0.5$, $K = 100$, $S(0) = 100$, $t = 12$, and the observation times contained in array **zeta**. We then create the statistical collector **statValue**, perform 10^5 simulation runs, and print the results. The discount factor e^{-rT} and the constants $\sigma\sqrt{\zeta_j - \zeta_{j-1}}$ and $(r - \sigma^2/2)(\zeta_j - \zeta_{j-1})$ are precomputed in the constructor **Asian**, to speed up the simulation.

The program in Listing 12 extends the class **Asian** to **AsianQMC**, whose method **simulate-QMC** estimates the option value via randomized quasi-Monte Carlo. It uses m independently randomized copies of digital net **p** and puts the results in statistical collector **statAverage**. The randomization is a left matrix scramble followed by a digital random shift, applied before each batch of n simulation runs.

Listing 12: Pricing an Asian option on a GMB process with quasi-Monte Carlo

```
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.hups.*;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.Chrono;

public class AsianQMC extends Asian {

   public AsianQMC (double r, double sigma, double strike,
                    double s0, int s, double[] zeta) {
       super (r, sigma, strike, s0, s, zeta);
   }

   // Makes m independent randomizations of the digital net p using stream
   // noise. For each of them, performs one simulation run for each point
   // of p, and adds the average over these points to the collector statQMC.
   public void simulateQMC (int m, DigitalNet p,
                            RandomStream noise, Tally statQMC) {
      Tally statValue  = new Tally ("stat on value of Asian option");
      PointSetIterator stream = p.iterator ();
      for (int j=0; j<m; j++) {
          p.leftMatrixScramble (noise);
          p.addRandomShift (0, p.getDimension(), noise);
          stream.resetStartStream();
          simulateRuns (p.getNumPoints(), stream, statValue);
          statQMC.add (statValue.average());
      }
   }


   public static void main (String[] args) {
      int s = 12;
      double[] zeta = new double[s+1];
      for (int j=0; j<=s; j++)
         zeta[j] = (double)j / (double)s;
      AsianQMC process = new AsianQMC (0.05, 0.5, 100.0, 100.0, s, zeta);
      Tally statValue  = new Tally ("value of Asian option");
      Tally statQMC = new Tally ("QMC averages for Asian option");

      Chrono timer = new Chrono();
      int n = 100000;
      System.out.println ("Ordinary MC:\n");
      process.simulateRuns (n, new MRG32k3a(), statValue);
      statValue.setConfidenceIntervalStudent();
      System.out.println (statValue.report (0.95, 3));
      System.out.println ("Total CPU time: " + timer.format());
      double varMC = statValue.variance();
      double cpuMC = timer.getSeconds() / n;  // CPU seconds per run.
      System.out.println ("------------------------\n");

      timer.init();
      DigitalNet p = new SobolSequence (16, 31, s); // 2^{16} points.
      n = p.getNumPoints();
      int m = 20;                           // Number of QMC randomizations.
      process.simulateQMC (m, p, new MRG32k3a(), statQMC);
      System.out.println ("QMC with Sobol point set with " + n +
          " points and affine matrix scramble:\n");
      statQMC.setConfidenceIntervalStudent();
      System.out.println (statQMC.report (0.95, 3));
      System.out.println ("Total CPU time: " + timer.format() + "\n");
      double varQMC = p.getNumPoints() * statQMC.variance();
      double cpuQMC = timer.getSeconds() / (m * n);
      System.out.printf ("Variance ratio:   %9.4g%n", varMC/varQMC);
      System.out.printf ("Efficiency ratio: %9.4g%n",
           (varMC * cpuMC) / (varQMC * cpuQMC));
   }
}
```

The random number stream passed to the method `simulateRuns` is an iterator that enumerates the points and coordinates of the randomized point set `p`. These point set iterators, available for each type of point set in package `hups`, implement the `RandomStream` interface and permit one to easily replace the uniform random numbers by (randomized) highly-uniform point sets or sequences, without changing the code of the model itself. The method `resetStartStream`, invoked immediately after each randomization, resets the iterator to the first coordinate of the first point of the point set `p`. The number n of simulation runs is equal to the number of points. The points correspond to substreams in the `RandomStream` interface. The method `resetNextSubstream`, invoked after each simulation run in `simulateRuns`, resets the iterator to the first coordinate of the next point. Each generation of a uniform random number (directly or indirectly) with this stream during the simulation moves the iterator to the next coordinate of the current point.

The point set used in this example is a *Sobol' net* with $n = 2^{16}$ points in t dimensions. The `main` program passes this point set to `simulateQMC` and asks for $m = 20$ independent randomizations. It then computes the empirical variance and CPU time *per simulation run* for both MC and randomized QMC. It prints the ratio of variances, which can be interpreted as the estimated *variance reduction factor* obtained when using QMC instead of MC in this example, and the ratio of efficiencies, which can be interpreted as the estimated *efficiency improvement factor*. (The efficiency of an estimator is defined as 1/(variance $\times$ CPU time per run.) The results are in Listing 13: QMC reduces the variance by a factor of around 250 and improves the efficiency by a factor of over 500. Randomized QMC not only reduces the variance, it also runs faster than MC. The main reason for this is the call to `resetNextSubstream` in `simulateRuns`, which is rather costly for a random number stream of class `MRG32k3a` (with the current implementation) and takes negligible time for an iterator over a digital net in base 2. In fact, in the the case of MC, the call to `resetNextSubstream` is not really needed. Removing it for that case reduces the CPU time by more than 40%.

Listing 13: Results of the program `AsianQMC`

```
Ordinary MC:

REPORT on Tally stat. collector ==> value of Asian option
    num. obs.      min          max        average    standard dev.
    100000       0.000     386.378      13.119      22.696
  95.0% confidence interval for mean (student): (    12.978,    13.260 )

Total CPU time: 0:0:0.71
------------------------

QMC with Sobol point set with 65536 points and affine matrix scramble:

REPORT on Tally stat. collector ==> QMC averages for Asian option
    num. obs.      min          max        average    standard dev.
        20      13.108      13.133      13.120      5.6E-3
  95.0% confidence interval for mean (student): (    13.118,    13.123 )

Total CPU time: 0:0:3.43

Variance ratio:       250.2
Efficiency ratio:     678.8
```

4 Discrete-Event Simulation

Examples of discrete-event simulation programs, based on the event view supported by the package `simevents`, are given in this section.

4.1 The single-server queue with an event view

We return to the single-server queue considered in Section 3.4. This time, instead of simulating a fixed number of customers, we simulate the system for a fixed time horizon of 1000.

Listing 14: Event-oriented simulation of an $M/M/1$ queue

```java
import umontreal.iro.lecuyer.simevents.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;
import umontreal.iro.lecuyer.stat.*;
import java.util.LinkedList;

public class QueueEv {

   RandomVariateGen genArr;
   RandomVariateGen genServ;
   LinkedList<Customer> waitList = new LinkedList<Customer> ();
   LinkedList<Customer> servList = new LinkedList<Customer> ();
   Tally custWaits     = new Tally ("Waiting times");
   Accumulate totWait  = new Accumulate ("Size of queue");

   class Customer { double arrivTime, servTime; }

   public QueueEv (double lambda, double mu) {
      genArr = new ExponentialGen (new MRG32k3a(), lambda);
      genServ = new ExponentialGen (new MRG32k3a(), mu);
   }

   public void simulateOneRun (double timeHorizon) {
      Sim.init();
      new EndOfSim().schedule (timeHorizon);
      new Arrival().schedule (genArr.nextDouble());
      Sim.start();
   }

   class Arrival extends Event {
      public void actions() {
         new Arrival().schedule (genArr.nextDouble()); // Next arrival.
         Customer cust = new Customer();  // Cust just arrived.
         cust.arrivTime = Sim.time();
         cust.servTime = genServ.nextDouble();
         if (servList.size() > 0) {       // Must join the queue.
```

```
            waitList.addLast (cust);
            totWait.update (waitList.size());
         } else {                             // Starts service.
            custWaits.add (0.0);
            servList.addLast (cust);
            new Departure().schedule (cust.servTime);
         }
      }
   }

   class Departure extends Event {
      public void actions() {
         servList.removeFirst();
         if (waitList.size() > 0) {
            // Starts service for next one in queue.
            Customer cust = waitList.removeFirst();
            totWait.update (waitList.size());
            custWaits.add (Sim.time() - cust.arrivTime);
            servList.addLast (cust);
            new Departure().schedule (cust.servTime);
         }
      }
   }

   class EndOfSim extends Event {
      public void actions() {
         Sim.stop();
      }
   }

   public static void main (String[] args) {
      QueueEv queue = new QueueEv (1.0, 2.0);
      queue.simulateOneRun (1000.0);
      System.out.println (queue.custWaits.report());
      System.out.println (queue.totWait.report());
   }
}
```

Listing 14 gives an event-oriented simulation program, where a subclass of the class `Event` is defined for each type of event that can occur in the simulation: arrival of a customer (`Arrival`), departure of a customer (`Departure`), and end of the simulation (`EndOfSim`). Each event *instance* is inserted into the *event list* upon its creation, with a scheduled time of occurrence, and is *executed* when the simulation clock reaches this time. Executing an event means invoking its `actions` method. Each event subclass must implement this method. The simulation clock and the event list (i.e., the list of events scheduled to occur in the future) are maintained behind the scenes by the class `Sim` of package `simevents`.

When `QueueEv` is instantiated by the `main` method, the program creates two streams of random numbers, two random variate generators, two lists, and two statistical probes (or collectors). The random number streams are attached to random variate generators `genArr` and `genServ` which are used to generate the times between successive arrivals and the service times, respectively. We can use such an attached generator because the means (parameters) do not change during simulation. The lists `waitList` and `servList` contain the customers waiting in the queue and the customer in service (if any), respectively. Maintaining a list for the customer in service may seem exaggerated, because this list never contains more than one object, but the current design has the advantage of working with very little change if the queuing model has more than one server, and in other more general situations. Note that we could have used the class `LinkedListStat` from package `simevents` instead of `java.util.LinkedList`. However, with our current implementation, the automatic statistical collection in that `LinkedListStat` class would not count the customers whose waiting time is zero, because they are never placed in the list.

The statistical probe `custWaits` collects statistics on the customer's waiting times. It is of the class `Tally`, which is appropriate when the statistical data of interest is a sequence of observations $X_1, X_2, \dots$ of which we might want to compute the sample mean, variance, and so on. A new observation is given to this probe by the `add` method each time a customer starts its service. Every `add` to a `Tally` probe brings a new observation X_i, which corresponds here to a customer's waiting time in the queue. The other statistical probe, `totWait`, is of the class `Accumulate`, which means that it computes the integral (and, eventually, the time-average) of a continuous-time stochastic process with piecewise-constant trajectory. Here, the stochastic process of interest is the length of the queue as a function of time. One must call `totWait.update` whenever there is a change in the queue size, to update the (hidden) *accumulator* that keeps the current value of the integral of the queue length. This integral is equal, after each update, to the total waiting time in the queue, for all the customers, since the beginning of the simulation.

Each customer is an object with two fields: `arrivTime` memorizes this customer's arrival time to the system, and `servTime` memorizes its service time. This object is created, and its fields are initialized, when the customer arrives.

The method `simulateOneRun` simulates this system for a fixed time horizon. It first invokes `Sim.init`, which initializes the clock and the event list. The method `Sim.start` actually starts the simulation by advancing the clock to the time of the first event in the event list, removing this event from the list, and executing it. This is repeated until either `Sim.stop` is called or the event list becomes empty. `Sim.time` returns the current time on the simulation clock. Here, two events are scheduled before starting the simulation: the end of the simulation at time horizon, and the arrival of the first customer at a random time that has the exponential distribution with *rate* λ (i.e., *mean* $1/\lambda$), generated by `genArr` using inversion and its attached random stream. The method `genArr.nextDouble` returns this exponential random variate.

The method `actions` of the class `Arrival` describes what happens when an arrival occurs. Arrivals are scheduled by a domino effect: the first action of each arrival event schedules the next event in a random number of time units, generated from the exponential distribution with rate λ. Then, the newly arrived customer is created, its arrival time is set to the current simulation time, and its service time is generated from the exponential distribution with mean $1/\mu$, using the random variate generator `genServ`. If the server is busy, this customer is inserted at the end of the queue (the list `waitList`) and the statistical probe `totWait`, that keeps track of the size of the queue, is updated. Otherwise, the customer is inserted in the server's list `servList`, its departure is scheduled to happen in a number of time units equal to its service time, and a new observation of 0.0 is given to the statistical probe `custWaits` that collects the waiting times.

When a `Departure` event occurs, the customer in service is removed from the list (and disappears). If the queue is not empty, the first customer is removed from the queue (`waitList`) and inserted in the server's list, and its departure is scheduled. The waiting time of that customer (the current time minus its arrival time) is given as a new observation to the probe `custWaits`, and the probe `totWait` is also updated with the new (reduced) size of the queue.

The event `EndOfSim` stops the simulation. Then the `main` routine regains control and prints statistical reports for the two probes. The results are shown in Listing 15. When calling `report` on an `Accumulate` object, an implicit update is done using the current simulation time and the last value given to `update`. In this example, this ensures that the `totWait` accumulator will integrate the total wait until the time horizon, because the simulation clock is still at that time when the report is printed. Without such an automatic update, the accumulator would integrate only up to the last update time before the time horizon.

Listing 15: Results of the program `QueueEv`

```
REPORT on Tally stat. collector ==> Waiting times
    num. obs.     min         max        average    standard dev.
      1037       0.000       6.262       0.495        0.835

REPORT on Accumulate stat. collector ==> Size of queue
      from time    to time       min         max          average
        0.00       1000.00      0.000      10.000          0.513
```

4.2 Continuous simulation: A prey-predator system

We consider a classical prey-predator system, where the preys are food for the predators (see, e.g., [8], page 87). Let $x(t)$ and $z(t)$ be the numbers of preys and predators at time t, respectively. These numbers are integers, but as an approximation, we shall assume that they are real-valued variables evolving according to the differential equations

$$\begin{aligned} x'(t) &= rx(t) - cx(t)z(t) \\ z'(t) &= -sz(t) + dx(t)z(t) \end{aligned}$$

with initial values $x(0) = x_0 > 0$ et $z(0) = z_0 > 0$. This is a *Lotka-Volterra* system of differential equations, which has a known analytical solution. Here, in the program of Listing 16, we simply simulate its evolution, to illustrate the continuous simulation facilities of SSJ.

Listing 16: Simulation of the prey-predator system

```
import umontreal.iro.lecuyer.simevents.*;

public class PreyPred {
   double r  = 0.005,      c  = 0.00001,
          s  = 0.01,       d  = 0.000005,     h = 5.0;
   double x0 = 2000.0,     z0 = 150.0;
   double horizon = 501.0;
   Simulator sim = new Simulator();
   Continuous x;
   Continuous z;

   public static void main (String[] args) { new PreyPred(); }

   public PreyPred() {
      x = new Preys(sim);
      z = new Preds(sim);
      sim.init();
      new EndOfSim(sim).schedule (horizon);
      new PrintPoint(sim).schedule (h);
      (sim.continuousState()).selectRungeKutta4 (h);
      x.startInteg (x0);
      z.startInteg (z0);
      sim.start();
   }

   public class Preys extends Continuous {
      public Preys(Simulator sim) { super(sim); }

      public double derivative (double t) {
         return (r * value() - c * value() * z.value());
      }
   }

   public class Preds extends Continuous {
      public Preds(Simulator sim) { super(sim); }

      public double derivative (double t) {
         return (-s * value() + d * x.value() * value());
      }
   }

   class PrintPoint extends Event {
      public PrintPoint(Simulator sim) { super(sim); }
      public void actions() {
         System.out.println (sim.time() + "  " + x.value() + "  " + z.value());
         this.schedule (h);
      }
   }

   class EndOfSim extends Event {
      public EndOfSim(Simulator sim) { super(sim); }
      public void actions() { sim.stop(); }
   }
}
```

Note that, instead of using the default simulator, this program explicitly creates a discrete-event `Simulator` object to manage the execution of the simulation, unlike the other examples in this section.

The program prints the triples $(t, x(t), z(t))$ at values of t that are multiples of `h`, one triple per line. This is done by an event of class `PrintPoint`, which is rescheduled at every `h` units of time. This output can be redirected to a file for later use, for example to plot a graph of the trajectory. The continuous variables `x` and `z` are instances of the classes `Preys` and `Preds`, whose method `derivative` give their derivative $x'(t)$ and $z'(t)$, respectively. The differential equations are integrated by a Runge-Kutta method of order 4.

4.3 A simplified bank

This is Example 1.4.1 of [2], page 14. A bank has a random number of tellers every morning. On any given day, the bank has t tellers with probability q_t, where $q_3 = 0.80$, $q_2 = 0.15$, and $q_1 = 0.05$. All the tellers are assumed to be identical from the modeling viewpoint.

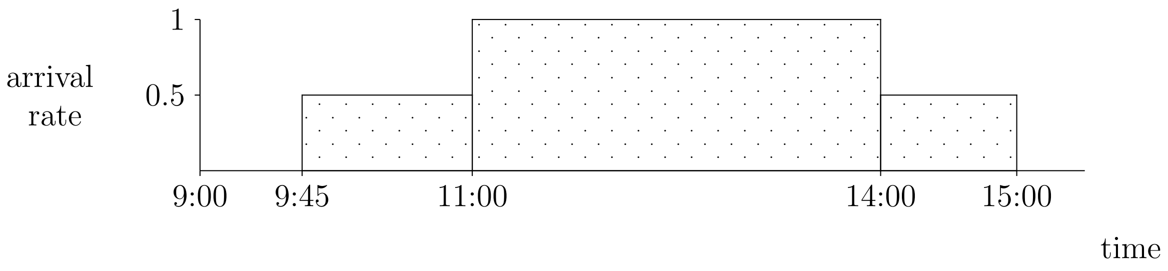


Figure 1: Arrival rate of customers to the bank.

Listing 17: Event-oriented simulation of the bank model

```
import umontreal.iro.lecuyer.simevents.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;
import umontreal.iro.lecuyer.stat.*;

public class BankEv {
   static final double minute = 1.0 / 60.0;
```

```
int      nbTellers;         // Number of tellers.
int      nbBusy;            // Number of tellers busy.
int      nbWait;            // Queue length.
int      nbServed;          // Number of customers served so far
double   meanDelay;         // Mean time between arrivals.
Event    nextArriv       = new Arrival();   // The next arrival.
RandomStream  streamArr  = new MRG32k3a();   // Customer's arrivals
ErlangGen genServ = new ErlangConvolutionGen (new MRG32k3a(), 2, 1.0/minute);
RandomStream  streamTeller = new MRG32k3a(); // Number of tellers
RandomStream  streamBalk   = new MRG32k3a(); // Balking decisions
Tally statServed = new Tally ("Nb. served per day");
Tally avWait     = new Tally ("Average wait per day (hours)");
Accumulate wait  = new Accumulate ("cumulated wait for this day");

Event e9h45 = new Event() {
   public void actions() {
      meanDelay = 2.0*minute;
      nextArriv.schedule
         (ExponentialGen.nextDouble (streamArr, 1.0/meanDelay));
   }
};

Event e10h = new Event() {
   public void actions() {
      double u = streamTeller.nextDouble();
      if (u >= 0.2) nbTellers = 3;
      else if (u < 0.05) nbTellers = 1;
      else nbTellers = 2;
      while (nbWait > 0 && nbBusy < nbTellers) {
         nbBusy++;  nbWait--;
         new Departure().schedule (genServ.nextDouble());
      }
      wait.update (nbWait);
   }
};

Event e11h = new Event() {
   public void actions() {
      nextArriv.reschedule ((nextArriv.time() - Sim.time())/2.0);
      meanDelay = minute;
   }
};

Event e14h = new Event() {
   public void actions() {
      nextArriv.reschedule ((nextArriv.time() - Sim.time())*2.0);
      meanDelay = 2.0*minute;
   }
};

Event e15h = new Event() {
   public void actions() { nextArriv.cancel(); }
};

private boolean balk() {
   return (nbWait > 9) ||
          (nbWait > 5 && (5.0*streamBalk.nextDouble() < nbWait-5));
}

class Arrival extends Event {
   public void actions() {
      nextArriv.schedule
         (ExponentialGen.nextDouble (streamArr, 1.0/meanDelay));
      if (nbBusy < nbTellers) {
         nbBusy++;
         new Departure().schedule (genServ.nextDouble());
      } else if (!balk())
         { nbWait++;  wait.update (nbWait); }
   }
}

class Departure extends  Event {
   public void actions() {
      nbServed++;
      if (nbWait > 0) {
         new Departure().schedule (genServ.nextDouble());
         nbWait--;   wait.update (nbWait);
      }
      else nbBusy--;
   }
};

public void simulOneDay() {
   Sim.init();       wait.init();
   nbTellers = 0;    nbBusy    = 0;
   nbWait    = 0;    nbServed  = 0;
   e9h45.schedule (9.75);
   e10h.schedule (10.0);
   e11h.schedule (11.0);
   e14h.schedule (14.0);
   e15h.schedule (15.0);
   Sim.start();
   statServed.add (nbServed);
   wait.update();
```

```
      avWait.add (wait.sum());
   }

   public void simulateDays (int numDays) {
      for (int i=1; i<=numDays; i++)  simulOneDay();
      System.out.println (statServed.report());
      System.out.println (avWait.report());
   }

   public static void main (String[] args) {
       new BankEv().simulateDays (100);
   }
}
```

The bank opens at 10:00 and closes at 15:00 (i.e., 3 P.M.). The customers arrive randomly according to a Poisson process with piecewise constant rate $\lambda(t)$, $t \geq 0$. The arrival rate $\lambda(t)$ (see Fig. 1) is 0.5 customer per minute from 9:45 until 11:00 and from 14:00 until 15:00, and one customer per minute from 11:00 until 14:00. The customers who arrive between 9:45 and 10:00 join a FIFO queue and wait for the bank to open. At 15:00, the door is closed, but all the customers already in will be served. Service starts at 10:00.

Customers form a FIFO queue for the tellers, with balking. An arriving customer will balk (walk out) with probability p_k if there are k customers ahead of him in the queue (not counting the people receiving service), where

$$p_k = \begin{cases} 0 & \text{if } k \leq 5; \\ (n-5)/5 & \text{if } 5 < k < 10; \\ 1 & \text{if } k \geq 10. \end{cases}$$

The customer service times are independent Erlang random variables: Each service time is the sum of two independent exponential random variables with mean one.

We want to estimate the expected number of customers served in a day, and the expected average wait for the customers served on a day.

Listing 17 gives and event-oriented simulation program for this bank model. There are events at the fixed times 9:45, 10:00, etc. At 9:45, the counters are initialized and the arrival process is started. The time until the first arrival, or the time between one arrival and the next one, is (tentatively) an exponential with a mean of 2 minutes. However, as soon as an arrival turns out to be past 11:00, its time must be readjusted to take into account the increase of the arrival rate at 11:00. The event 11:00 takes care of this readjustment, and the event at 14:00 makes a similar readjustment when the arrival rate decreases. We give the specific name `nextArriv` to the next planned arrival event in order to be able to reschedule that particular event to a different time. Note that a *single* arrival event is created at the beginning and this same event is scheduled over and over again. This can be done because there is never more than one arrival event in the event list. (We could have done that as well for the $M/M/1$ queue in Listing 14.)

At the bank opening at 10:00, an event generates the number of tellers and starts the service for the corresponding customers. The event at 15:00 cancels the next arrival.

Upon arrival, a customer checks if a teller is free. If so, one teller becomes busy and the customer generates its service time and schedules his departure, otherwise the customer either balks or joins the queue. The balking decision is computed by the method `balk`, using the random number stream `streamBalk`. The arrival event also generates the next scheduled arrival. Upon departure, the customer frees the teller, and the first customer in the queue, if any, can start its service. The generator `genServ` is an `ErlangConvolutionGen` generator, so that the Erlang variates are generated by adding two exponentials instead of using inversion.

The method `simulateDays` simulates the bank for `numDays` days and prints a statistical report. If X_i is the number of customers served on day i and Q_i the total waiting time on day i, the program estimates $E[X_i]$ and $E[Q_i]$ by their sample averages $\bar{X}_n$ and $\bar{Q}_n$ with $n =$`numDays`. For each simulation run (each day), `simulOneDay` initializes the clock, event list, and statistical probe for the waiting times, schedules the deterministic events, and runs the simulation. After 15:00, no more arrival occurs and the event list becomes empty when the last customer departs. At that point, the program returns to right after the `Sim.start()` statement and updates the statistical counters for the number of customers served during the day and their total waiting time.

The results are given in Listing 18.

Listing 18: Results of the `BankEv` program

```
REPORT on Tally stat. collector ==> Nb. served per day
    num. obs.      min          max        average    standard dev.
       100      152.000      285.000      240.590       19.210

REPORT on Tally stat. collector ==> Average wait per day (hours)
    num. obs.      min          max        average    standard dev.
       100        0.816       35.613        4.793        5.186
```

4.4 A call center

We consider here a simplified model of a telephone contact center (or *call center*) where agents answer incoming calls. Each day, the center operates for m hours. The number of agents answering calls and the arrival rate of calls vary during the day; we shall assume that they are constant within each hour of operation but depend on the hour. Let n_j be the number of agents in the center during hour j, for $j = 0, \ldots, m-1$. For example, if the center operates from 8 AM to 9 PM, then $m = 13$ and hour j starts at $(j+8)$ o'clock. All agents are assumed to be identical. When the number of occupied agents at the end of hour j is larger than n_{j+1}, ongoing calls are all completed but new calls are answered only when there are less than n_{j+1} agents busy. After the center closes, ongoing calls are completed and calls already in the queue are answered, but no additional incoming call is taken.

The calls arrive according to a Poisson process with piecewise constant rate, equal to $R_j = B\lambda_j$ during hour j, where the λ_j are constants and B is a random variable having the gamma distribution with parameters (α_0, α_0). Thus, B has mean 1 and variance $1/\alpha_0$, and it represents the *busyness* of the day; it is more busy than usual when $B > 1$ and less busy when $B < 1$. The Poisson process assumption means that conditional on B, the number of incoming calls during any subinterval $(t_1, t_2]$ of hour j is a Poisson random variable with mean $(t_2 - t_1)B\lambda_j$ and that the arrival counts in any disjoint time intervals are independent random variables. This arrival process model is motivated and studied in [15] and [1].

Incoming calls form a FIFO queue for the agents. A call is *lost* (abandons the queue) when its waiting time exceed its *patience time*. The patience times of calls are assumed to be i.i.d. random variables with the following distribution: with probability p the patience time is 0 (so the person hangs up unless there is an agent available immediately), and with probability $1-p$ it is exponential with mean $1/\nu$. The service times are i.i.d. gamma random variables with parameters (α, β).

We want to estimate the following quantities *in the long run* (i.e., over an infinite number of days): (a) w, the average waiting time per call, (b) $g(s)$, the fraction of calls whose waiting time is less than s seconds for a given threshold s, and (c) ℓ, the fraction of calls lost due to abandonment.

Suppose we simulate the model for n days. For each day i, let A_i be the number of arrivals, W_i the total waiting time of all calls, $G_i(s)$ the number of calls who waited less than s seconds, and L_i the number of abandonments. For this model, the expected number of incoming calls in a day is $a = E[A_i] = \sum_{j=0}^{m-1} \lambda_j$. Then, W_i/a, $G_i(s)/a$, and L_i/a, $i = 1, \ldots, n$, are i.i.d. unbiased estimators of w, $g(s)$, and ℓ, respectively, and can be used to compute confidence intervals for these quantities in a standard way if n is large.

Listing 19: Simulation of a simplified call center

```java
import umontreal.iro.lecuyer.simevents.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;
import umontreal.iro.lecuyer.probdist.*;
import umontreal.iro.lecuyer.stat.*;
import java.io.*;
import java.util.*;

public class CallCenter {
   static final double HOUR = 3600.0;  // Time is in seconds.

   // Data
   // Arrival rates are per hour, service and patience times are in seconds.
   double openingTime;    // Opening time of the center (in hours).
   int numPeriods;        // Number of working periods (hours) in the day.
   int[] numAgents;       // Number of agents for each period.
   double[] lambda;       // Base arrival rate lambda_j for each j.
   double alpha0;         // Parameter of gamma distribution for B.
   double p;               // Probability that patience time is 0.
   double nu;              // Parameter of exponential for patience time.
   double alpha, beta;     // Parameters of gamma service time distribution.
   double s;               // Want stats on waiting times smaller than s.

   // Variables
   double busyness;        // Current value of B.
   double arrRate = 0.0;   // Current arrival rate.
   int nAgents;            // Number of agents in current period.
   int nBusy;              // Number of agents occupied;
   int nArrivals;          // Number of arrivals today;
   int nAbandon;           // Number of abandonments during the day.
   int nGoodQoS;           // Number of waiting times less than s today.
   double nCallsExpected;  // Expected number of calls per day.

   Event nextArrival = new Arrival();            // The next Arrival event.
   LinkedList<Call> waitList = new LinkedList<Call>();

   RandomStream streamB         = new MRG32k3a(); // For B.
   RandomStream streamArr       = new MRG32k3a(); // For arrivals.
   RandomStream streamPatience = new MRG32k3a(); // For patience times.
   GammaGen genServ;       // For service times; created in readData().

   Tally[] allTal = new Tally [4];
   Tally statArrivals = allTal[0] = new Tally ("Number of arrivals per day");
   Tally statWaits = allTal[1] = new Tally ("Average waiting time per customer");
   Tally statGoodQoS = allTal[2] = new Tally ("Proportion of waiting times < s");
   Tally statAbandon = allTal[3] = new Tally ("Proportion of calls lost");
   Tally statWaitsDay = new Tally ("Waiting times within a day");

   public CallCenter (String fileName) throws IOException {
      readData (fileName);
      // genServ can be created only after its parameters are read.
      // The acceptance/rejection method is much faster than inversion.
      genServ = new GammaAcceptanceRejectionGen (new MRG32k3a(), alpha, beta);
   }

   // Reads data and construct arrays.
   public void readData (String fileName) throws IOException {
      Locale loc = Locale.getDefault();
      Locale.setDefault(Locale.US); // to read reals as 8.3 instead of 8,3
      BufferedReader input = new BufferedReader (new FileReader (fileName));
      Scanner scan = new Scanner(input);
      openingTime = scan.nextDouble();      scan.nextLine();
      numPeriods = scan.nextInt();          scan.nextLine();
      numAgents = new int[numPeriods];
      lambda = new double[numPeriods];
```

```
      nCallsExpected = 0.0;
      for (int j = 0; j < numPeriods; j++) {
         numAgents[j] = scan.nextInt();
         lambda[j] = scan.nextDouble();
         nCallsExpected += lambda[j];       scan.nextLine();
      }
      alpha0 = scan.nextDouble();      scan.nextLine();
      p = scan.nextDouble();           scan.nextLine();
      nu = scan.nextDouble();          scan.nextLine();
      alpha = scan.nextDouble();       scan.nextLine();
      beta = scan.nextDouble();        scan.nextLine();
      s = scan.nextDouble();
      scan.close();
      Locale.setDefault(loc);
   }

   // A phone call.
   class Call {
      double arrivalTime, serviceTime, patienceTime;

      public Call() {
         serviceTime = genServ.nextDouble(); // Generate service time.
         if (nBusy < nAgents) {           // Start service immediately.
            nBusy++;
            nGoodQoS++;
            statWaitsDay.add (0.0);
            new CallCompletion().schedule (serviceTime);
         } else {                         // Join the queue.
            patienceTime = generPatience();
            arrivalTime = Sim.time();
            waitList.addLast (this);
         }
      }

      public void endWait() {
         double wait = Sim.time() - arrivalTime;
         if (patienceTime < wait) { // Caller has abandoned.
            nAbandon++;
            wait = patienceTime;    // Effective waiting time.
         } else {
            nBusy++;
            new CallCompletion().schedule (serviceTime);
         }
         if (wait < s) nGoodQoS++;
         statWaitsDay.add (wait);
      }
   }
```

```
// Event: A new period begins.
class NextPeriod extends Event {
   int j;     // Number of the new period.
   public NextPeriod (int period) { j = period; }
   public void actions() {
      if (j < numPeriods) {
         nAgents = numAgents[j];
         arrRate = busyness * lambda[j] / HOUR;
         if (j == 0) {
            nextArrival.schedule
               (ExponentialDist.inverseF (arrRate, streamArr.nextDouble()));
         } else {
            checkQueue();
            nextArrival.reschedule ((nextArrival.time() - Sim.time())
                                    * lambda[j-1] / lambda[j]);
         }
         new NextPeriod(j+1).schedule (1.0 * HOUR);
      } else
         nextArrival.cancel();  // End of the day.
   }
}

// Event: A call arrives.
class Arrival extends Event {
   public void actions() {
      nextArrival.schedule
         (ExponentialDist.inverseF (arrRate, streamArr.nextDouble()));
      nArrivals++;
      new Call();               // Call just arrived.
   }
}

// Event: A call is completed.
class CallCompletion extends Event {
   public void actions() { nBusy--;   checkQueue(); }
}

// Start answering new calls if agents are free and queue not empty.
public void checkQueue() {
   while ((waitList.size() > 0) && (nBusy < nAgents))
      (waitList.removeFirst()).endWait();
}

// Generates the patience time for a call.
public double generPatience() {
   double u = streamPatience.nextDouble();
```

```
      if (u <= p)
         return 0.0;
      else
         return ExponentialDist.inverseF (nu, (1.0-u) / (1.0-p));
   }

   public void simulateOneDay (double busyness) {
      Sim.init();        statWaitsDay.init();
      nArrivals = 0;     nAbandon = 0;
      nGoodQoS = 0;      nBusy = 0;
      this.busyness = busyness;

      new NextPeriod(0).schedule (openingTime * HOUR);
      Sim.start();
      // Here the simulation is running...

      statArrivals.add ((double)nArrivals);
      statAbandon.add ((double)nAbandon / nCallsExpected);
      statGoodQoS.add ((double)nGoodQoS / nCallsExpected);
      statWaits.add (statWaitsDay.sum() / nCallsExpected);
   }

   public void simulateOneDay () {
      simulateOneDay (GammaDist.inverseF (alpha0, alpha0, 8,
                                          streamB.nextDouble()));
   }

   static public void main (String[] args) throws IOException {
      CallCenter cc = new CallCenter ("CallCenter.dat");
      for (int i = 0; i < 1000; i++)  cc.simulateOneDay();
      System.out.println ("\nNum. calls expected = " + cc.nCallsExpected +"\n");
      for (int i = 0; i < cc.allTal.length; i++) {
         cc.allTal[i].setConfidenceIntervalStudent();
         cc.allTal[i].setConfidenceLevel (0.90);
      }
      System.out.println (Tally.report ("CallCenter:", cc.allTal));
   }
}
```

Listing 19 gives an event-oriented simulation program for this call center model. When the `CallCenter` class is instantiated by the `main` method, the random streams, list, and statistical probes are created, and the model parameters are read from a file by the method `readData`. The line `Locale.setDefault(Locale.US)` is added because real numbers in the data file are read in the anglo-saxon form 8.3 instead of the form 8,3 used by most countries in the world. The `main` program then simulates $n = 1000$ operating days and prints the value of a, as well as 90% confidence intervals on a, w, $g(s)$, and ℓ, based on their estimators $\bar{A}_n$,

$\bar{W}_n/a$, $\bar{G}_n(s)/a$, and $\bar{L}_n/a$, assuming that these estimators have approximately the Student distribution. This is justified by the fact that W_i, and $G_i(s)$, and L_i are themselves "averages" over several observations, so we may expect their distribution to be not far from a normal.

To generate the service times, we use a gamma random variate generator called `genServ`, created in the constructor after the parameters (α, β) of the service time distribution have been read from the data file. For the other random variables in the model, we simply create random streams of i.i.d. uniforms (in the preamble) and apply inversion explicitly to generate the random variates. The latter approach is more convenient, e.g., for patience times because their distribution is not standard and for the inter-arrival times because their mean changes every period. For the gamma service time distribution, on the other hand, the parameters always remain the same and inversion is rather slow, so we decided to create a generator that uses a faster special method.

The method `simulateOneDay` simulates one day of operation. It initializes the simulation clock, event list, and counters, schedules the center's opening and the first arrival, and starts the simulation. When the day is over, it updates the statistical collectors. Note that there are two versions of this method; one that generates the random variate B and the other that takes its value as an input parameter. This is convenient in case one wishes to simulate the center with a fixed value of B.

An event `NextPeriod(j)` marks the beginning of each period j. The first of these events (for $j = 0$) is scheduled by `simulateOneDay`; then the following ones schedule each other successively, until the end of the day. This type of event updates the number of agents in the center and the arrival rate for the next period. If the number of agents has just increased and the queue is not empty, some calls in the queue can now be answered. The method `checkQueue` verifies this and starts service for the appropriate number of calls. The time until the next planned arrival is readjusted to take into account the change of arrival rate, as follows. The inter-arrival times are i.i.d. exponential with mean $1/R_{j-1}$ when the arrival rate is fixed at R_{j-1}. But when the arrival rate changes from R_{j-1} to R_j, the residual time until the next arrival should be modified from an exponential with mean $1/R_{j-1}$ (already generated) to an exponential with mean $1/R_j$. Multiplying the residual time by λ_{j-1}/λ_j is an easy way to achieve this. We give the specific name `nextArrival` to the next arrival event in order to be able to reschedule it to a different time. Note that there is a *single* arrival event which is scheduled over and over again during the entire simulation. This is more efficient than creating a new arrival event for each call, and can be done here because there is never more than one arrival event at a time in the event list. At the end of the day, simply canceling the next arrival makes sure that no more calls will arrive.

Each arrival event first schedules the next one. Then it increments the arrivals counter and creates the new call that just arrived. The call's constructor generates its service time and decides where the incoming call should go. If an agent is available, the call is answered immediately (its waiting time is zero), and an event is scheduled for the completion of the call. Otherwise, the call must join the queue; its patience time is generated by `generPatience` and memorized, together with its arrival time, for future reference.

Upon completion of a call, the number of busy agents is decremented and one must verify if a waiting call can now be answered. The method `checkQueue` verifies that and if

Listing 20: Simulation of a simplified call center

```
Num. calls expected = 1660.0

Report for CallCenter:
    num obs.     min         max         average      std. dev.   conf. int. 90.0%

Num. arrivals per day:
    1000       460.000    4206.000     1639.507       513.189   ( 1612.8, 1666.2)
Average wait time per customer:
    1000         0.000     570.757       11.834        34.078   ( 10.060, 13.608)
Proportion of waiting times < s:
    1000         0.277       1.155        0.853         0.169   ( 0.844,  0.862)
Proportion of calls lost:
    1000         0.000       0.844        0.034         0.061   ( 0.031,  0.037)
```

the answer is yes, it removes the first call from the queue and activates its `endWait` method. This method first compares the call's waiting time with its patience time, to see if this call is still waiting or has been lost (by abandonment). If the call was lost, we consider its waiting time as being equal to its patience time (i.e., the time that the caller has really waited), for the statistics. If the call is still there, the number of busy agents is incremented and an event is scheduled for the call completion.

The results of this program, with the data in file `CallCenter.dat`, are shown in Listing 20.

This model is certainly an oversimplification of actual call centers. It can be embellished and made more realistic by considering different types of agents, different types of calls, agents taking breaks for lunch, coffee, or going to the restroom, agents making outbound calls to reach customers when the inbound traffic is low (e.g., for marketing purpose or for returning calls), and so on. One could also model the revenue generated by calls and the operating costs for running the center, and use the simulation model to compare alternative operating strategies in terms of the expected net revenue, for example.

5 Process-Oriented Programs

The process-oriented programs discussed in this section are based on the `simprocs` package and were initially designed to run using *Java green threads*, which are actually *simulated* threads and have been available only in JDK versions 1.3.1 or earlier, unfortunately. The green threads in these early versions of JDK were obtained by running the program with the "`java -classic`" option.

With *native threads*, these programs run more slowly and may crash if too many threads are initiated. This is a serious limitation of the package `simprocs`, due to the fact that Java threads are *not* designed for simulation.

A second implementation of `simprocs`, available in DSOL (see the class `DSOLProcess-Simulator`) and provided to us by Peter Jacobs, does not use the Java threads. It is based on a Java reflection mechanism that interprets the code of processes at runtime and transforms everything into events. A program using the process view and implemented with the DSOL interpreter can be 500 to 1000 times slower than the corresponding event-based program. On the other hand, it is a good and safe teaching tool for process-oriented programming, and the number of processes is limited only by the available memory. To use the DSOL system with a program that imports `simprocs`, it suffices to run the program with the "`java -Dssj.withDSOL ...`" option.

5.1 The single queue

Typical simulation languages offer higher-level constructs than those used in the program of Listing 14, and so does SSJ. This is illustrated by our second implementation of the single-server queue model, in Listing 21, based on a paradigm called the *process-oriented* approach.

In the event-oriented implementation, each customer was a *passive* object, storing two real numbers, and performing no action by itself. In the process-oriented implementation given in Listing 21, each customer (instance of the class `Customer`) is a *process* whose activities are described by its `actions` method. This is implemented by associating a Java `Thread` to each `SimProcess`. The server is an object of the class `Resource`, created when `QueueProc` is instantiated by `main`. It is a *passive* object, in the sense that it executes no code. Active resources, when needed, can be implemented as processes.

When it starts executing its actions, a customer first schedules the arrival of the next customer, as in the event-oriented case. (Behind the scenes, this effectively schedules an event, in the event list, that will start a new customer instance. The class `SimProcess` contains all scheduling facilities of `Event`, which permits one to schedule processes just like events.) The customer then requests the server by invoking `server.request`. If the server is free, the customer gets it and can continue its execution immediately. Otherwise, the customer is automatically (behind the scenes) placed in the server's queue, is suspended, and resumes its execution only when it obtains the server. When its service can start, the customer invokes `delay` to freeze itself for a duration equal to its service time, which is

Listing 21: Process-oriented simulation of an $M/M/1$ queue

```java
import umontreal.iro.lecuyer.simevents.*;
import umontreal.iro.lecuyer.simprocs.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;
import umontreal.iro.lecuyer.stat.*;

public class QueueProc {
   Resource server = new Resource (1, "Server");
   RandomVariateGen genArr;
   RandomVariateGen genServ;

   public QueueProc (double lambda, double mu) {
      genArr = new ExponentialGen (new MRG32k3a(), lambda);
      genServ = new ExponentialGen (new MRG32k3a(), mu);
   }

   public void simulateOneRun (double timeHorizon) {
      SimProcess.init();
      server.setStatCollecting (true);
      new EndOfSim().schedule (timeHorizon);
      new Customer().schedule (genArr.nextDouble());
      Sim.start();
   }

   class Customer extends SimProcess {
      public void actions() {
         new Customer().schedule (genArr.nextDouble());
         server.request (1);
         delay (genServ.nextDouble());
         server.release (1);
      }
   }

   class EndOfSim extends Event {
      public void actions() { Sim.stop(); }
   }

   public static void main (String[] args) {
      QueueProc queue = new QueueProc (1.0, 2.0);
      queue.simulateOneRun (1000.0);
      System.out.println (queue.server.report());
   }
}
```

again generated from the exponential distribution with mean $1/\mu$ using the random variate generator `genServ`. After this delay has elapsed, the customer releases the server and ends its life. Invoking `delay(d)` can be interpreted as scheduling an event that will resume the execution of the process in `d` units of time. Note that several distinct customers can co-exist in the simulation at any given point in time, and be at different phases of their `actions` method. However, only one process is executing at a time.

The constructor `QueueProc` initializes the simulation, invokes `setStatCollecting (true)` to specify that detailed statistical collection must be performed automatically for the resource `server`, schedules an event `EndOfSim` at the time horizon, schedules the first customer's arrival, and starts the simulation. The `EndOfSim` event stops the simulation. The `main` program then regains control and prints a detailed statistical report on the resource `server`.

It should be pointed out that in the `QueueProc` program, the service time of a customer is generated only when the customer starts its service, whereas for `QueueEv`, it was generated at customer's arrival. For this particular model, it turns out that this makes no difference in the results, because the customers are served in a FIFO order and because one random number stream is dedicated to the generation of service times. However, this may have an impact in other situations.

The process-oriented program here is more compact and more elegant than its event-oriented counterpart. This tends to be often true: Process-oriented programming frequently gives less cumbersome and better looking programs. On the other hand, the process-oriented implementations also tend to execute more slowly, because they involve more overhead. For example, the process-driven single-server queue simulation is two to three times slower than its event-driven counterpart. In fact, process management is done via the event list: processes are started, suspended, reactivated, and so on, by hidden events. If the execution speed of a simulation program is really important, it may be better to stick to an event-oriented implementation.

Listing 22: Results of the program `QueueProc`

```
REPORT ON RESOURCE : Server
   From time :    0.00   to time :    1000.00
                    min         max      average  standard dev.  nb. obs.
   Capacity           1           1        1.000
   Utilization        0           1        0.530
   Queue Size         0          10        0.513
   Wait           0.000       6.262        0.495      0.835       1037
   Service       6.5E-4       3.437        0.511      0.512       1037
   Sojourn       8.2E-4       6.466        1.005      0.979       1037
```

Listing 22 shows the output of the program `QueueProc`. It contains more information than the output of `QueueEv`. It gives statistics on the server utilization, queue size, waiting times, service times, and sojourn times in the system. (The sojourn time of a customer is its waiting time plus its service time.)

We see that by time $T = 1000$, 1037 customers have completed their waiting and all of them have completed their service. The maximal queue size has been 10 and its average length between time 0 and time 1000 was 0.513. The waiting times were in the range from 0 to 6.262, with an average of 0.495, while the service times were from 0.00065 to 3.437, with an average of 0.511 (recall that the theoretical mean service time is $1/\mu = 0.5$). Clearly, the largest waiting time and largest service time belong to different customers.

The report also gives the empirical standard deviations of the waiting, service, and sojourn times. It is important to note that these standard deviations should *not* be used to compute confidence intervals for the expected average waiting times or sojourn times in the standard way, because the observations here (e.g., the successive waiting times) are strongly dependent, and also not identically distributed. Appropriate techniques for computing confidence intervals in this type of situation are described, e.g., in [3, 8].

5.2 A job shop model

This example is adapted from [8, Section 2.6], and from [9]. A job shop contains M groups of machines, the mth group having s_m identical machines, for $m = 1, \dots, M$. It is modeled as a network of queues: each group has a single FIFO queue, with s_m identical servers for the mth group. There are N types of tasks arriving to the shop at random. Tasks of type n arrive according to a Poisson process with rate λ_n per hour, for $n = 1, \dots, N$. Each type of task requires a fixed sequence of operations, where each operation must be performed on a specific type of machine and has a deterministic duration. A task of type n requires p_n operations, to be performed on machines $m_{n,1}, m_{n,2}, \dots, m_{n,p_n}$, in that order, and whose respective durations are $d_{n,1}, d_{n,2}, \dots, d_{n,p_n}$, in hours. A task can pass more than once on the same machine type, so p_n may exceed M.

We want to simulate the job shop for T hours, assuming that it is initially empty, and start collecting statistics only after a warm-up period of T_0 hours. We want to compute: (a) the average sojourn time in the shop for each type of task and (b) the average utilization rate, average length of the waiting queue, and average waiting time, for each type of machine, over the time interval $[T_0, T]$. For the average sojourn times and waiting times, the counted observations are the sojourn times and waits that *end* during the time interval $[T_0, T]$. Note that the only randomness in this model is in the task arrival process.

The class `Jobshop` in Listing 23 performs this simulation. Each group of machine is viewed as a resource, with capacity s_m for the group m. The different *types* of task are objects of the class `TaskType`. This class is used to store the parameters of the different types: their arrival rate, the number of operations, the machine type and duration for each operation, and a statistical collector for their sojourn times in the shop. (In the program, the machine types and task types are numbered from 0 to $M - 1$ and from 0 to $N - 1$, respectively, because array indices in Java start at 0.)

The tasks that circulate in the shop are objects of the class `Task`. The `actions` method in class `Task` describes the behavior of a task from its arrival until it exits the shop. Each task, upon arrival, schedules the arrival of the next task of the same type. The task then runs through the list of its operations. For each operation, it requests the appropriate type of machine, keeps it for the duration of the operation, and releases it. When the task terminates, it sends its sojourn time as a new observation to the collector `statSojourn`.

Before starting the simulation, the method `simulateOneRun` schedules an event for the end of the simulation and another one for the end of the warm-up period. The latter simply starts the statistical collection. It also schedules the arrival of a task of each type. Each task will in turn schedules the arrival of the next task of its own type.

With this implementation, the event list always contain N “task arrival” events, one for each type of task. An alternative implementation would be that each task schedules another task arrival in a number of hours that is an exponential r.v. with rate λ, where $\lambda = \lambda_0 + \cdots + \lambda_{N-1}$ is the global arrival rate, and then the type of each arriving task is n with probability λ_n/λ, independently of the others. Initially, a *single* arrival would be scheduled by the class `Jobshop`. This approach is stochastically equivalent to the current implementation (see, e.g., [2, 16]), but the event list contains only one “task arrival” event at a time. On the other hand, there is the additional work of generating the task type on each arrival.

At the end of the simulation, the `main` program prints the statistical reports. Note that if one wanted to perform several independent simulation runs with this program, the statistical collectors would have to be reinitialized before each run, and additional collectors would be required to collect the run averages and compute confidence intervals.

Listing 23: A job shop simulation

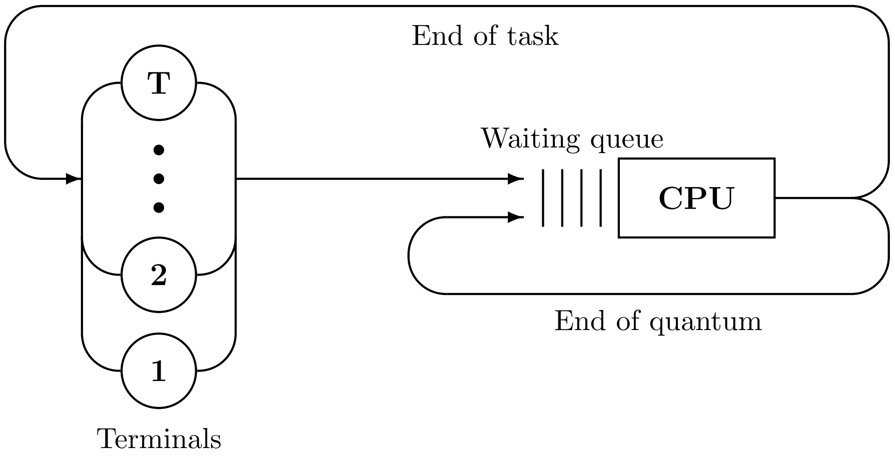
```
import umontreal.iro.lecuyer.simevents.*;
import umontreal.iro.lecuyer.simprocs.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.ExponentialGen;
import umontreal.iro.lecuyer.stat.Tally;
import java.io.*;
import java.util.*;

public class Jobshop {
   int nbMachTypes;       // Number of machine types M.
   int nbTaskTypes;       // Number of task types N.
   double warmupTime;     // Warmup time T_0.
   double horizonTime;    // Horizon length T.
   boolean warmupDone;    // Becomes true when warmup time is over.
   Resource[] machType;   // The machines groups as resources.
   TaskType[] taskType;   // The task types.
   RandomStream streamArr = new MRG32k3a(); // Stream for arrivals.
   BufferedReader input;

   public Jobshop() throws IOException { readData(); }

   // Reads data file, and creates machine types and task types.
   void readData() throws IOException {
      input = new BufferedReader (new FileReader ("Jobshop.dat"));
      StringTokenizer line = new StringTokenizer (input.readLine());
      warmupTime = Double.parseDouble (line.nextToken());
      line = new StringTokenizer (input.readLine());
      horizonTime = Double.parseDouble (line.nextToken());
      line = new StringTokenizer (input.readLine());
      nbMachTypes = Integer.parseInt (line.nextToken());
      nbTaskTypes = Integer.parseInt (line.nextToken());
      machType = new Resource[nbMachTypes];
      for (int m=0; m < nbMachTypes; m++) {
         line = new StringTokenizer (input.readLine());
         String name = line.nextToken();
         int nb = Integer.parseInt (line.nextToken());
         machType[m] = new Resource (nb, name);
      }
      taskType = new TaskType[nbTaskTypes];
      for (int n=0; n < nbTaskTypes; n++)
         taskType[n] = new TaskType();
      input.close();
   }


   class TaskType {
      public String     name;        // Task name.
      public double     arrivalRate; // Arrival rate.
      public int        nbOper;      // Number of operations.
      public Resource[] machOper;    // Machines where operations occur.
      public double[]   lengthOper;  // Durations of operations.
      public Tally      statSojourn; // Stats on sojourn times.

      // Reads data for new task type and creates data structures.
      TaskType() throws IOException {
         StringTokenizer line = new StringTokenizer (input.readLine());
         statSojourn = new Tally (name = line.nextToken());
         arrivalRate = Double.parseDouble (line.nextToken());
         nbOper = Integer.parseInt (line.nextToken());
         machOper = new Resource[nbOper];
         lengthOper = new double[nbOper];
         for (int i = 0; i < nbOper; i++) {
            int p = Integer.parseInt (line.nextToken());
            machOper[i] = machType[p-1];
            lengthOper[i] = Double.parseDouble (line.nextToken());
         }
      }

      // Performs the operations of this task (to be called by a process).
      public void performTask (SimProcess p) {
         double arrivalTime = Sim.time();
         for (int i=0; i < nbOper; i++) {
            machOper[i].request (1); p.delay (lengthOper[i]);
            machOper[i].release (1);
         }
         if (warmupDone) statSojourn.add (Sim.time() - arrivalTime);
      }
   }

   public class Task extends SimProcess {
      TaskType type;

      Task (TaskType type) { this.type = type; }

      public void actions() {
          // First schedules next task of this type, then executes task.
         new Task (type).schedule (ExponentialGen.nextDouble
               (streamArr, type.arrivalRate));
         type.performTask (this);
      }
   }

   Event endWarmup = new Event() {
      public void actions() {
         for (int m=0; m < nbMachTypes; m++)
            machType[m].setStatCollecting (true);
         warmupDone = true;
      }
   };

   Event endOfSim = new Event() {
      public void actions() { Sim.stop(); }
   };

   public void simulateOneRun() {
      SimProcess.init();
      endOfSim.schedule (horizonTime);
      endWarmup.schedule (warmupTime);
      warmupDone = false;
      for (int n = 0; n < nbTaskTypes; n++) {
         new Task (taskType[n]).schedule (ExponentialGen.nextDouble
            (streamArr, taskType[n].arrivalRate));
      }
      Sim.start();
   }

   public void printReportOneRun() {
      for (int m=0; m < nbMachTypes; m++)
         System.out.println (machType[m].report());
      for (int n=0; n < nbTaskTypes; n++)
         System.out.println (taskType[n].statSojourn.report());
   }

   static public void main (String[] args) throws IOException {
      Jobshop shop = new Jobshop();
      shop.simulateOneRun();
      shop.printReportOneRun();
   }
}
```

5.3 A time-shared computer system

This example is adapted from [8], Section 2.4. Consider a simplified time-shared computer system comprised of T identical and independent terminals, all busy, using a common server (e.g., for database requests, or central processing unit (CPU) consumption, etc.). Each terminal user sends a task to the server at some random time and waits for the response. After receiving the response, he thinks for some random time before submitting a new task, and so on.

We assume that the thinking time is an exponential random variable with mean μ, whereas the server's time needed for a request is a Weibull random variable with parameters α, λ and δ. The tasks waiting for the server form a single queue with a *round robin* service policy with *quantum size* q, which operates as follows. When a task obtains the server, if it can be completed in less than q seconds, then it keeps the server until completion. Otherwise, it gets the server for q seconds and returns to the back of the queue to be continued later. In both cases, there is also h additional seconds of *overhead* for changing the task that has the server's attention.

The *response time* of a task is defined as the difference between the time when the task ends (including the overhead h at the end) and the arrival time of the task to the server. We are interested in the *mean response time*, in steady-state. We will simulate the system until N tasks have ended, with all terminals initially in the "thinking" state. To reduce the initial bias, we will start collecting statistics only after N_0 tasks have ended (so the first N_0 response times are not counted by our estimator, and we take the average response time for the $N - N_0$ response times that remain). This entire simulation is repeated R times, independently, so we can estimate the variance of our estimator.



Suppose we want to compare the mean response times for two different configurations of this system, where a configuration is characterized by the vector of parameters $(T, q, h, \mu, \alpha, \lambda, \delta)$. We will make R independent simulation runs (replications) for each configuration. To compare the two configurations, we want to use *common random numbers*, i.e., the same streams of random numbers across the two configurations. We couple the simulation runs by pairs: for run number i, let R_{1i} and R_{2i} be the mean response times for configurations 1 and 2, and let

$$D_i = R_{1i} - R_{2i}.$$

We use the same random numbers to obtain R_{1i} and R_{2i}, for each i. The D_i are nevertheless independent random variables (under the blunt assumption that the random streams really produce independent uniform random variables) and we can use them to compute a confidence interval for the difference d between the theoretical mean response times of the two systems. Using common random numbers across R_{1i} and R_{2i} should reduce the variance of the D_i and the size of the confidence interval.

Listing 24: Simulation of a time shared system

```
import umontreal.iro.lecuyer.simevents.*;
import umontreal.iro.lecuyer.simprocs.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;
import umontreal.iro.lecuyer.stat.Tally;
import java.io.*;

public class TimeShared {
   int nbTerminal    = 20;     // Number of terminals.
   double quantum;             // Quantum size.
   double overhead  = 0.001; // Amount of overhead (h).
   double meanThink = 5.0;   // Mean thinking time.
   double alpha     = 0.5;   // Parameters of the Weibull service times.
   double lambda    = 1.0;   //                ''
   double delta     = 0.0;   //                ''
   int N            = 1100;  // Total number of tasks to simulate.
   int N0           = 100;   // Number of tasks for warmup.
   int nbTasks;              // Number of tasks ended so far.

   RandomStream streamThink  = new MRG32k3a();
   RandomVariateGen genThink = new ExponentialGen (streamThink, 1.0/meanThink);
   RandomStream streamServ   = new MRG32k3a ("Gen. for service requirements");
   RandomVariateGen genServ = new WeibullGen (streamServ, alpha, lambda, delta);
   Resource server        = new Resource (1, "The server");
   Tally meanInRep        = new Tally ("Average for current run");
   Tally statDiff         = new Tally ("Diff. on mean response times");

   class Terminal extends SimProcess {
      public void actions() {
         double arrivTime;    // Arrival time of current request.
         double timeNeeded;   // Server's time still needed for it.
         while (nbTasks < N) {
            delay (genThink.nextDouble());
            arrivTime = Sim.time();
            timeNeeded = genServ.nextDouble();
            while (timeNeeded > quantum) {
               server.request (1);
               delay (quantum + overhead);
               timeNeeded -= quantum;
               server.release (1);
            }
            server.request (1);  // Here, timeNeeded <= quantum.
            delay (timeNeeded + overhead);
            server.release (1);
            nbTasks++;
            if (nbTasks > N0)
               meanInRep.add (Sim.time() - arrivTime);
                       // Take the observation if warmup is over.
         }
         Sim.stop();            // N tasks have now completed.
      }
   }

   private void simulOneRun() {
      SimProcess.init();
      server.init();
      meanInRep.init();
      nbTasks = 0;
      for (int i=1; i <= nbTerminal; i++)
         new Terminal().schedule (0.0);
      Sim.start();
   }

   // Simulate numRuns pairs of runs and prints a confidence interval
   // on the difference of perf. for quantum sizes q1 and q2.
   public void simulateConfigs (double numRuns, double q1, double q2) {
      double mean1;  // To memorize average for first configuration.
      for (int rep = 0; rep < numRuns; rep++) {
         quantum = q1;
         simulOneRun();
         mean1 = meanInRep.average();
         streamThink.resetStartSubstream();
         streamServ.resetStartSubstream();
         quantum = q2;
         simulOneRun();
         statDiff.add (mean1 - meanInRep.average());
         streamThink.resetNextSubstream();
         streamServ.resetNextSubstream();
      }
      statDiff.setConfidenceIntervalStudent();
      System.out.println (statDiff.report (0.9, 3));
   }

   public static void main (String[] args) {
      new TimeShared().simulateConfigs (10, 0.1, 0.2);
   }
}
```

The program of Listing 24 performs this simulation. In the `simulateConfigs` method, we perform `numRuns` pairs of simulation runs, one with quantum size `q1` and one with quantum size q2. Each random stream is reset to the beginning of its *current* substream after the first run, and to the beginning of its *next* substream after the second run, to make sure that for each pair of runs, the generators start from exactly the same seeds and generate exactly the same random numbers in the same order. The variable `mean1` memorizes the values of R_{1i} after the first run, and the statistical probe `statDiff` collects the differences D_i, in order to compute a confidence interval for d.

For each simulation run, the statistical probe `meanInRep` is used to compute the average response time for the $N - N_0$ tasks that terminate after the warm-up. It is initialized before each run and updated with a new observation at the ith task termination, for $i = N_0 + 1, \ldots, N$. At the beginning of a run, a `Terminal` process is activated for each terminal. When the Nth task terminates, the corresponding process invokes `Sim.stop` to stop the simulation and to return the control to the instruction that follows the call to `simulOneRun`.

For a concrete example, let $T = 20$, $h = .001$, $\mu = 5$ sec., $\alpha = 1/2$, $\lambda = 1$ and $\delta = 0$ for the two configurations. With these parameters, the mean of the Weibull distribution is 2. Take $q = 0.1$ for configuration 1 and $q = 0.2$ for configuration 2. We also choose $N_0 = 100$, $N = 1100$, and $R = 10$ runs. With these numbers, the program gives the results of Listing 25. The confidence interval on the difference between the response time with $q = 0.1$ and that with $q = 0.2$ contains only positive numbers. We can therefore conclude that the mean response time is significantly shorter (statistically) with $q = 0.2$ than with $q = 0.1$ (assuming that we can neglect the bias due to the choice of the initial state). To gain better confidence in this conclusion, we could repeat the simulation with larger values of N_0 and N.

Listing 25: Difference in mean response times for $q = 0.1$ and $q = 0.2$, for time shared system

```
REPORT on Tally stat. collector ==> Diff. on mean response times
    num. obs.      min          max        average     standard dev.
        10      -0.134       0.369        0.168        0.174
  90.0% confidence interval for mean (student): (     0.067,     0.269 )
```

Of course, the model could be made more realistic by considering, for example, different types of terminals, with different parameters, a number of terminals that changes with time, different classes of tasks with priorities, etc. SSJ offers the tools to implement these generalizations easily.

5.4 Guided visits

This example is translated from [9]. A touristic attraction offers guided visits, using three guides. The site opens at 10:00 and closes at 16:00. Visitors arrive in small groups (e.g., families) and the arrival process of those groups is assumed to be a Poisson process with rate of 20 groups per hour, from 9:45 until 16:00. The visitors arriving before 10:00 must wait for the opening. After 16:00, the visits already under way can be completed, but no new visit is undertaken, so that all the visitors still waiting cannot visit the site and are lost.

The size of each arriving group of visitors is a discrete random variable taking the value i with probability p_i given in the following table:

i	1	2	3	4
p_i	.2	.6	.1	.1

Visits are normally made by groups of 8 to 15 visitors. Each visit requires one guide and lasts 45 minutes. People waiting for guides form a single queue. When a guide becomes free, if there is less than 8 people in the queue, the guide waits until the queue grows to at least 8 people, otherwise she starts a new visit right away. If the queue contains more than 15 people, the first 15 will go on this visit. At 16:00, if there is less than 8 people in the queue and a guide is free, she starts a visit with the remaining people. At noon, each free guide takes 30 minutes for lunch. The guides that are busy at that time will take 30 minutes for lunch as soon as they complete their on-going visit.

Sometimes, an arriving group of visitors may decide to just go away (balk) because the queue is too long. We assume that the probability of balking when the queue size is n is given by

$$R(n) = \begin{cases} 0 & \text{for } n \le 10; \\ (n-10)/30 & \text{for } 10 < n < 40; \\ 1 & \text{for } n \ge 40. \end{cases}$$

The aim is to estimate the average number of visitors lost per day, in the long run. The visitors lost are those that balk or are still in the queue after 16:00.

A simulation program for this model is given in Listing 26. Here, time is measured in hours, starting at midnight. At time 9:45, for example, the simulation clock is at 9.75. The (process) class `Guide` describes the daily behavior of a guide (each guide is an instance of this class), whereas `Arrival` generates the arrivals according to a Poisson process, the group sizes, and the balking decisions. The event `closing` closes the site at 16:00.

The `Bin` mechanism `visitReady` is used to synchronize the `Guide` processes. The number of tokens in this bin is 1 if there are enough visitors in the queue to start a visit (8 or more) and is 0 otherwise. When the queue size reaches 8 due to a new arrival, the `Arrival` process puts a token into the bin. This wakes up a guide if one is free. A guide must take a token from the bin to start a new visit. If there are still 8 people or more in the queue when she starts the visit, she puts the token back to the bin immediately, to indicate that another visit is ready to be undertaken by the next available guide.

Listing 26: Simulation of guided visits

```
import umontreal.iro.lecuyer.simevents.*;
import umontreal.iro.lecuyer.simprocs.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;
import umontreal.iro.lecuyer.stat.*;
// import umontreal.iro.lecuyer.simprocs.dsol.SimProcess;

public class Visits {
   int queueSize;     // Size of waiting queue.
   int nbLost;        // Number of visitors lost so far today.
   Bin visitReady = new Bin ("Visit ready");
                      // A token becomes available when there
                      // are enough visitors to start a new visit.
   Tally avLost = new Tally ("Nb. of visitors lost per day");
   RandomVariateGen genArriv = new ExponentialGen (new MRG32k3a(), 20.0);
                                             // Interarriv.
   RandomStream  streamSize  = new MRG32k3a(); // Group sizes.
   RandomStream  streamBalk  = new MRG32k3a(); // Balking decisions.

   private void oneDay() {
      queueSize = 0;   nbLost = 0;
      SimProcess.init();
      visitReady.init();
      closing.schedule (16.0);
      new Arrival().schedule (9.75);
      for (int i=1; i<=3; i++) new Guide().schedule (10.0);
      Sim.start();
      avLost.add (nbLost);
   }

   Event closing = new Event() {
      public void actions() {
         if (visitReady.waitList().size() == 0)
            nbLost += queueSize;
         Sim.stop();
      }
   };

   class Guide extends SimProcess {
      public void actions() {
         boolean lunchDone = false;
         while (true) {
            if (Sim.time() > 12.0 && !lunchDone) {
               delay (0.5);  lunchDone = true;
            }
            visitReady.take (1);  // Starts the next visit.
            if (queueSize > 15) queueSize -= 15;
            else queueSize = 0;
            if (queueSize >= 8) visitReady.put (1);
                                 // Enough people for another visit.
            delay (0.75);
         }
      }
   }

   class Arrival extends SimProcess {
      public void actions() {
         while (true) {
            delay (genArriv.nextDouble());
            // A new group of visitors arrives.
            int groupSize;  // number of visitors in group.
            double u = streamSize.nextDouble();
            if (u <= 0.2)      groupSize = 1;
            else if (u <= 0.8) groupSize = 2;
            else if (u <= 0.9) groupSize = 3;
            else               groupSize = 4;
            if (!balk()) {
               queueSize += groupSize;
               if (queueSize >= 8 &&
                   visitReady.getAvailable() == 0)
                  // A token is now available.
                  visitReady.put (1);
            }
            else  nbLost += groupSize;
         }
      }

      private boolean balk() {
         if (queueSize <= 10)  return false;
         if (queueSize >= 40)  return true;
         return (streamBalk.nextDouble() <
         ((queueSize - 10.0) / 30.0));
      }
   }

   public void simulateRuns (int numRuns) {
      for (int i = 1; i <= numRuns; i++) oneDay();
      avLost.setConfidenceIntervalStudent();
      System.out.println (avLost.report (0.9, 3));
   }

   static public void main (String[] args) {
      new Visits().simulateRuns (100);
   }
}
```

The simulation results are in Listing 27.

Listing 27: Simulation results for the guided visits model

```
REPORT on Tally stat. collector ==> Nb. of visitors lost per day
    num. obs.     min          max       average    standard dev.
       100       3.000      48.000      21.780      10.639
  90.0% confidence interval for mean (student): (    20.014,    23.546 )
```

Could we have used a `Condition` instead of a `Bin` for synchronizing the `Guide` processes? The problem would be that if several guides are waiting for a condition indicating that the queue size has reached 8, *all* these guides (not only the first one) would resume their execution simultaneously when the condition becomes true.

5.5 Return to the simplified bank

Listing 28 gives a process-oriented simulation program for the bank model of Section 4.3. Here, the customers are viewed as processes. There are still events at the fixed times 9:45, 10:00, etc., and these events do the same thing, but they are implicit in the process-oriented implementation. The next planned arrival event `nextArriv` is replaced by the next planned arriving customer `nextCust`.

Instead of using the default process simulator as in the previous examples, here one creates a `ThreadProcessSimulator` object that will manage the processes of this simulation.

The process-oriented version of the program is shorter, because certain aspects (such as the details of an arrival or departure event) are taken care of automatically by the process/resource construct, and the events 9:45, 10:00, etc., are replaced by a single process. At 10 o'clock, the `setCapacity` statement that fixes the number of tellers also takes care of starting service for the appropriate number of customers. The two programs produce exactly the same results, given in Listing 17. However, the process-oriented program take approximately 4 to 5 times longer to run than its event-oriented counterpart.

Listing 28: Process-oriented simulation of the bank model

```
import umontreal.iro.lecuyer.simprocs.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;
import umontreal.iro.lecuyer.stat.*;

public class BankProc {

   ProcessSimulator sim = new ThreadProcessSimulator();
   double   minute = 1.0 / 60.0;
   int      nbServed;           // Number of customers served so far
   double   meanDelay;          // Mean time between arrivals
   SimProcess  nextCust;        // Next customer to arrive
   Resource tellers          = new Resource (sim, 0, "The tellers");
   RandomStream  streamArr   = new MRG32k3a(); // Customer's arrivals
   ErlangGen genServ         = new ErlangConvolutionGen
                                  (new MRG32k3a(), 2, 1.0/minute);
   RandomStream  streamTeller  = new MRG32k3a(); // Number of tellers
   RandomStream  streamBalk    = new MRG32k3a(); // Balking decisions
   Tally    statServed     = new Tally ("Nb. served per day");
   Tally    avWait         = new Tally ("Average wait per day (hours)");

   class OneDay extends SimProcess {
      public OneDay(ProcessSimulator sim) { super(sim); }
      public void actions() {
         int nbTellers;         // Number of tellers today.
         nbServed = 0;
         tellers.setCapacity (0);
         tellers.waitList().initStat();
         meanDelay = 2.0 * minute;
         // It is 9:45, start arrival process.
         (nextCust = new Customer(sim)).schedule
           (ExponentialGen.nextDouble (streamArr, 1.0/meanDelay));
         delay (15.0 * minute);
         // Bank opens at 10:00, generate number of tellers.
         double u = streamTeller.nextDouble();
         if (u >= 0.2) nbTellers = 3;
         else if (u < 0.05)  nbTellers = 1;
         else nbTellers = 2;
         tellers.setCapacity (nbTellers);
         delay (1.0);   // It is 11:00, double arrival rate.
         nextCust.reschedule (nextCust.getDelay() / 2.0);
         meanDelay = minute;
         delay (3.0);   // It is 14:00, halve arrival rate.
         nextCust.reschedule (nextCust.getDelay() * 2.0);
         meanDelay = 2.0 * minute;
         delay (1.0);   // It is 15:00, bank closes.
         nextCust.cancel();
      }
   }

   class Customer extends SimProcess {
      public Customer(ProcessSimulator sim) { super(sim); }
      public void actions() {
         (nextCust = new Customer(sim)).schedule
            (ExponentialGen.nextDouble (streamArr, 1.0/meanDelay));
         if (!balk()) {
            tellers.request (1);
            delay (genServ.nextDouble());
            tellers.release (1);
            nbServed++;
         }
      }

      private boolean balk() {
         int n = tellers.waitList().size();
         return n > 9 || (n > 5 && 5.0*streamBalk.nextDouble() < n - 5);
      }
   }

   public void simulOneDay() {
      sim.init();
      new OneDay(sim).schedule (9.75);
      sim.start();
      statServed.add (nbServed);
      avWait.add (tellers.waitList().statSize().sum());
   }

   public void simulateDays (int numDays) {
      tellers.waitList().setStatCollecting (true);
      for (int i=1; i<=numDays; i++)  simulOneDay();
      System.out.println (statServed.report());
      System.out.println (avWait.report());
   }
```

```
    public static void main (String[] args) {
        new BankProc().simulateDays (100);
    }
}
```

References

[1] A. N. Avramidis, A. Deslauriers, and P. L'Ecuyer. Modeling daily arrivals to a telephone call center. *Management Science*, 50(7):896–908, 2004.

[2] P. Bratley, B. L. Fox, and L. E. Schrage. *A Guide to Simulation.* Springer-Verlag, New York, NY, second edition, 1987.

[3] G. S. Fishman. *Monte Carlo: Concepts, Algorithms, and Applications.* Springer Series in Operations Research. Springer-Verlag, New York, NY, 1996.

[4] P. Glasserman. *Monte Carlo Methods in Financial Engineering.* Springer-Verlag, New York, 2004.

[5] Wolfgang Hoschek. *The Colt Distribution: Open Source Libraries for High Performance Scientific and Technical Computing in Java.* CERN, Geneva, 2004. Available at `http://dsd.lbl.gov/~hoschek/colt/`.

[6] J. J. Moré and B. S. Garbow and K. E. Hillstrom. *User Guide for MINPACK-1, Report ANL-80-74.* Argonne, Illinois, USA, 1980. See `http://www-fp.mcs.anl.gov/otc/Guide/softwareGuide/Blurbs/minpack.html`.

[7] L. Kleinrock. *Queueing Systems, Vol. 1.* Wiley, New York, NY, 1975.

[8] A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis.* McGraw-Hill, New York, NY, third edition, 2000.

[9] P. L'Ecuyer. SIMOD: Définition fonctionnelle et guide d'utilisation (version 2.0). Technical Report DIUL-RT-8804, Département d'informatique, Université Laval, Sept 1988.

[10] P. L'Ecuyer. *Stochastic Simulation.* 2008. Notes for a graduate simulation course, textbook in preparation.

[11] P. L'Ecuyer and E. Buist. Simulation in Java with SSJ. In M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, editors, *Proceedings of the 2005 Winter Simulation Conference*, pages 611–620, Pistacaway, NJ, 2005. IEEE Press.

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

[13] J. Leydold and W. Hörmann. *UNURAN—A Library for Universal Non-Uniform Random Number Generators*, 2002. Available at `http://statistik.wu-wien.ac.at/unuran`.

[14] R. B. Schnabel. *UNCMIN—Unconstrained Optimization Package, FORTRAN.* University of Colorado at Boulder. See `http://www.ici.ro/camo/unconstr/uncmin.htm`.

[15] W. Whitt. Dynamic staffing in a telephone call center aiming to immediately answer all calls. *Operations Research Letters*, 24:205–212, 1999.

[16] R. W. Wolff. *Stochastic Modeling and the Theory of Queues.* Prentice-Hall, New York, NY, 1989.