rstream: Streams of Random Numbers for Stochastic Simulation

by Pierre L'Ecuyer & Josef Leydold

Requirements for random number generators

Simulation modeling is a very important tool for solving complex real world problems (Law and Kelton, 2000). Crucial steps in a simulation study are (Banks, 1998):

1. problem formulation and model conceptualization;
2. input data analysis;
3. run simulation using streams of (pseudo)random numbers;
4. output data analysis;
5. validation of the model.

R is an excellent tool for Steps 2 and 4 and it would be quite nice if it would provide better support for Step 3 as well. This is of particular interest for investigating complex models that go beyond the standard ones that are implemented as templates in simulation software. One requirement for this purpose is to have good sources of (pseudo)random numbers available in the form of multiple streams that satisfy the following conditions:

- The underlying uniform random number generator must have excellent structural and statistical properties, see e.g. Knuth (1998) or L'Ecuyer (2004) for details.
- The streams must be reproducible. This is required for variance reduction techniques like common variables and antithetic variates, and for testing software.
- The state of a stream should be storable at any time.
- There should be the possibility to get different streams for different runs of the simulation ("seeding the random streams").
- The streams must be "statistically independent" to avoid unintended correlations between different parts of the model, or in parallel computing when each node runs its own random number generator.
- There should be substreams to keep simulations with common random numbers synchronized.
- It should be possible to rerun the entire simulation study with different sources of random numbers. This is necessary to detect possible (although extremely rare) incorrect results caused by interferences between the model and the chosen (kind of) random number generator. These different random number generators should share a common programming interface.

In R (like in most other simulation software or libraries for scientific computing) there is no concept of independent random streams. Instead, one global source of random numbers controlled by the global variables *.Random.seed*, which can be changed via `set.seed` and `RNGkind`, are available. The package **setRNG** tries to simplify the setting of this global random number generator. The functionalities listed above can only be mimicked by an appropriate sequence of `set.seed` calls together with proper management of state variables by the user. This is cumbersome.

Multiple streams can be created by jumping ahead by large numbers of steps in the sequence of a particular random number generator to start each new stream. Advancing the state by a large number of steps requires expertise (and even some investigation of the source code) that a user of a simulation software usually does not have. Just running the generator (without using the output) to jump ahead is too time consuming. Choosing random seeds for the different streams is too dangerous as there is some (although small) probability of overlap or strong correlation between the resulting streams.

A unified object-oriented interface for random streams

It is appropriate to treat random number streams as objects and to have methods to handle these streams and draw random samples. An example of this approach is **RngStreams** by L'Ecuyer et al. (2002). This library provides multiple independent streams and substreams, as well as antithetic variates. It is based on a combined multiple-recursive generator as the underlying "backbone" generator, whose very long period is split into many long independent streams viewed as objects These streams have substreams that can be accessed by a `nextsubstream` method.

We have designed a new interface to random number generators in R by means of S4 classes that treat random streams as objects. The interface is strongly influenced by **RngStreams**. It is implemented in the package **rstream**, available from CRAN. It consists of the following parts:

- Create an instance of an *rstream* object (*seed* is optional):[1]

  ```
  s <- new("rstream.mrg32k3a",
            seed=rep(12345,6))
  ```

  Consecutive calls of the constructor give "statistically independent" random streams. The destructor is called implicitly by the garbage collector.

- Draw a random sample (of size *n*):

  ```
  x <- rstream.sample(s)
  y <- rstream.sample(s,n=100)
  ```

- Reset the random stream;
  skip to the next substream:

  ```
  rstream.reset(s)
  rstream.nextsubstream(s)
  rstream.resetsubstream(s)
  ```

- Switch to antithetic numbers;
  use increased precision[2];
  read status of flags:

  ```
  rstream.antithetic(s) <- TRUE
  rstream.antithetic(s)
  rstream.incprecision(s) <- TRUE
  rstream.incprecision(s)
  ```

Notice that there is no need for seeding a *particular* random stream. There is a package seed for all streams that are instances of *rstream.mrg32k3a* objects. There is method `rstream.reset` for reseting a random stream. The state of the stream is stored in the object and setting seeds to obtain different streams is extremely difficult or dangerous (when the seeds are chosen at random).

Rstream objects store pointers to the state of the random stream. Thus, simply using the <- assignment creates two variables that point to the same stream (like copying an environment results in two variables that point to the same environment). Thus in parallel computing, *rstream* objects cannot be simply copied to a particular node of the computer cluster. Furthermore, *rstream* objects cannot be saved between R sessions. Thus we need additional methods to make independent copies (clones) of the same object and methods to save (pack) and restore (unpack) objects.

- Make an independent copy of the random stream (clone):

  ```
  sc <- rstream.clone(s)
  ```

- Save and restore a stream object (the packed rstream object can be stored and handled like any other R object):

  ```
  rstream.packed(s) <- TRUE
  rstream.packed(s) <- FALSE
  ```

Package **rstream** is designed to handle random number generators in a unified manner. Thus there is a class that deals with the R built-in generators:

- Create *rstream* object for built-in generator:

  ```
  s <- new("rstream.runif")
  ```

Additionally, it is easy to integrate other sources of random number generators (e.g. the generators from the GSL (http://www.gnu.org/software/gsl/), SPRNG (see also package **rsprng** on CRAN), or SSJ (http://www.iro.umontreal.ca/~simardr/ssj/)) into this framework which then can be used by the same methods. However, not all methods work for all types of generators. A method that fails responds with an error message.

The **rstream** package also interacts with the R random number generator, that is, the active global generator can be transformed into and handled as an *rstream* object and vice versa, every *rstream* object can be set as the global generator in R.

- Use stream *s* as global R uniform RNG:

  ```
  rstream.RNG(s)
  ```

- Store the status of the global R uniform RNG as *rstream* object:

  ```
  gs <- rstream.RNG()
  ```

Simple examples

We give elementary examples that illustrate how to use package **rstream**. The model considered in this section is quite simple and the estimated quantity could be computed with other methods (more accurately).

[1] '`rstream.mrg32k3a`' is the name of a particular class of random streams named after its underlying backbone generator. The library **RngStreams** by L'Ecuyer et al. (2002) uses the MRG32k3a multiple recursive generator. For other classes see the manual page of the package.

[2] By default the underlying random number generator used a resolution of 2^{-32} like the R built-in RNGs. This can be increased to 2^{-53} (the precision of the IEEE double format) by combining two consecutive random numbers. Notice that this is done implicitly in R when normal random variates are created via inversion (`RNGkind(normal.kind="Inversion")`), but not for other generation methods. However, it is more transparent when this behavior can be controlled by the user.

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.)

We implement this model with the following R code. (The implementation is kept as simple as possible. It is not optimized, uses global variables, and we have neglected any error handling.)

```
## load library
library(rstream)

## parameter for inventory system
lambda <- 100  # mean demand size
c      <- 2    # sales price
h      <- 0.1  # inventory cost per item
K      <- 10   # fixed ordering cost
k      <- 1    # marginal ordering cost per item
p      <- 0.95 # probability that order arrives
m      <- 200  # number of days
```

We use two independent sources of randomness for the simulation of the daily demand and for the success of ordering the product.

```
## initialize streams of random numbers
gendemand <- new("rstream.mrg32k3a")
genorder  <- new("rstream.mrg32k3a")
```

Notice that the following R code need not be changed at all if we replace `rstream.mrg32k3a` by a different source of random numbers that provides the required functionality.

For the simulation of daily sales we need a generator for (truncated) Poisson distributions. Currently a system for invoking generators for nonuniform distributions that use particular random number generators similar to the **rstream** package does not exist. Developing such a package is the next step. Meanwhile we use the following simple (and slow!) generator which is based on the inversion method (Devroye, 1986, §X.3.3).

```
## simple generator for Poisson distribution that
## uses uniform random numbers from stream 'rs'
randpoisson <- function (lambda,rs) {
   X <- 0
   sum <- exp(-lambda)
   prod <- exp(-lambda)
   U <- rstream.sample(rs,1)
   while (U > sum) {
      X <- X + 1
      prod <- prod * lambda / X
      sum <- sum + prod
   }
   return (X)
}
```

The last brickstone is a routine that computes a realization of the the average profit for a given control policy (s, S).

```
simulateOneRun <- function (s,S) {
   X <- S; profit <- 0
   for (j in 1:m) {
      sales <- randpoisson(lambda,gendemand)
      Y <- max(0,X-sales)
      profit <- 0
      if (Y < s && rstream.sample(genorder,1)<p) {
         profit <- profit - (K + k * (S-Y))
         X <- S }
      else {
         X <- Y }
   }
   return (profit/m)
}
```

Now we can perform 100 independent simulation runs for control policy $(s, S) = (80, 200)$ and compute the 95% confidence interval for the average daily sales.

```
result <- replicate(100,simulateOneRun(80,200))
t.test(result,conf.level=0.95)$conf.int

## confidence interval:
[1] 85.02457 85.43686
attr(,"conf.level")
[1] 0.95
```

Common random numbers

In the second example we want to compare the control policy $(s_0, S_0) = (80, 200)$ with policy $(s_1, S_1) = (80, 198)$ in our simple inventory model.

```
resultdiff <- replicate(100,
   simulateOneRun(80,200)-simulateOneRun(80,198) )
t.test(resultdiff,conf.level=0.95)$conf.int

## confidence interval:
[1] -0.3352277  0.2723377
attr(,"conf.level")
[1] 0.95
```

The variance of this estimator can be reduced by using the technique of common random numbers. We have to make sure that for both simulations the same sequence of random numbers is used in the same way. Thus we have to keep the random streams used for demand and ordering synchronized by means of reset calls.

```
## reset streams
rstream.reset(gendemand)
rstream.reset(genorder)

resultdiffCRN <- replicate(100,
     {
      ## skip to beginning of next substream
      rstream.nextsubstream(gendemand);
      rstream.nextsubstream(genorder);
      simulateOneRun(80,200)}
  -  {
      ## reset to beginning of current substream
      rstream.resetsubstream(gendemand);
      rstream.resetsubstream(genorder);
      simulateOneRun(80,198)} )

t.test(resultdiffCRN,conf.level=0.95)$conf.int

## confidence interval:
[1] 0.2354436 0.3717864
attr(,"conf.level")
[1] 0.95
```

The confidence interval is much narrower now than with independent random numbers.

Parallel computing

When stochastic simulations are run in parallel, it is crucial that the random numbers generated on each of the nodes in the computing environment are independent. The class `rstream.mrg32k3a` is perfectly suited for this task in a master/slave design.

```
## Master node
## create independent streams for each node
stream1 <- new("rstream.mrg32k3a")
rstream.packed(stream1) <- TRUE
stream2 <- new("rstream.mrg32k3a")
rstream.packed(stream2) <- TRUE

## Slave node
rstream.packed(stream1) <- FALSE
X <- rstream.sample(stream1,1);
```

Remark

Notice, that this "negative functionality" of un/packing the *rstream* object should be hidden in some user interface for parallel computing, e.g., **snow** (Simple Network of Workstations). This un/packing mechanism could be avoided when the state of the random stream is always stored inside R (instead of using external pointers). Package **rlecuyer** (version 0.1) implements access to the **RngStreams** library in such a way. However, we found out that this approach is rather slow. On our computer (Intel Pentium 4, 2.6 GHz, Linux 2.6.12, R version 2.1.1) the marginal generation time for a single random number is 10 times slower than with our approach (and 250 times slower for generating a random vector of size 100).

Adding uniform random number generators in R

In our opinion the proposed interface can serve as a brickstone of simulation packages based on the R environment. With this design, uniform random number generators can be used as arguments to subroutines. Thus random number generators can be easily exchanged or used in parallel. An extension for nonuniform random numbers is the next step towards a flexible and easy-to-use simulation environment in R.

There is one drawback in the implementation. R uses a static table to implement access to different kinds of random number generators. Only one slot (`RNGkind(kind="user-supplied")`) has been provided for users who want to add her own generator. This slot relies on a pointer to a function called `user_unif_rand`. Thus **rstream** has to abuse this slot to add additional random number generators. However, there are other packages on CRAN that use the same trick (**randaes**, **rlecuyer**, **rsprng**, **SuppDists**) or load one of these packages (**Rmpi**, **rpvm**, **rpart**, **snow**, **varSelRF**). Thus if a user loads two of these packages or adds her own uniform random number generator, the behavior of at least one of the packages is broken. This problem could be fixed by some changes in the R core system, either by adding a *package* variable to `RNGkind(kind="user-supplied")` similar to the `.Call` function, or by replacing the static table by a dynamic one.

Acknowledgments

This work was supported by the Austrian *Fonds zur Förderung der Wissenschaftlichen Forschung*, Proj.No. P16767-N12 and the Natural Sciences and Engineering Research Council of Canada, Grant Number ODGP0110050.

Bibliography

J. Banks, editor. *Handbook of Simulation*. Wiley, New York, 1998. 16

L. Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, New-York, 1986. 18

D. E. Knuth. *The Art of Computer Programming. Vol. 2: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1998. 16

A. M. Law and W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, 3rd edition, 2000. 16

P. L'Ecuyer. Random number generation. In J. E. Gentle, W. Haerdle, and Y. Mori, editors, *Handbook of Computational Statistics*, chapter II.2, pages 35–70. Springer-Verlag, Berrlin, 2004. 16

P. L'Ecuyer, R. Simard, E. J. Chen, and W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002. 16, 17

Pierre L'Ecuyer
Université de Montréal, Canada
lecuyer@iro.umontreal.ca

Josef Leydold
Wirtschaftsuniversität Wien, Austria
leydold@statistik.wu-wien.ac.at

mfp

Multivariable Fractional Polynomials

by Axel Benner

Introduction

The mfp package is targeted at the use of multivariable fractional polynomials for modelling the influence of continuous, categorical and binary covariates on the outcome in regression models, as introduced by Royston and Altman (1994) and modified by Sauerbrei and Royston (1999). It combines backward elimination with a systematic search for a 'suitable' transformation to represent the influence of each continuous covariate on the outcome. The stability of the models selected was investigated in Royston and Sauerbrei (2003). Briefly, fractional polynomials models are useful when one wishes to preserve the continuous nature of the covariates in a regression model, but suspects that some or all of the relationships may be non-linear. At each step of a 'backfitting' algorithm mfp constructs a fractional polynomial transformation for each continuous covariate while fixing the current functional forms of the other covariates. The algorithm terminates when no covariate has to be eliminated and the functional forms of the continuous covariates do not change anymore.

In regression models often decisions on the selection and the functional form of covariates must be taken. Although other choices are possible, we propose to use P-values to select models. With many covariates in the model, a large nominal P-value will lead to substantial overfitting. Most often a nominal P-value of 5% is used for variable selection.

The choice of a 'selection' P-value depends on the number of covariates and whether hypothesis generation is an important aim of the study. The mfp procedure therefore allows to distinguish between covariates of main interest and confounders by specifying different selection levels for different covariates.

Fractional Polynomials

Suppose that we have an outcome variable, a single continuous covariate, Z, and a suitable regression model relating them. Our starting point is the straight line model, $\beta_1 Z$ (for easier notation we ignore the intercept, β_0). Often this assumption is an adequate description of the functional relationship, but other functional forms must be investigated for possible improvements in model fit. A simple extension of the straight line is a power transformation model, $\beta_1 Z^p$. This model has often been used by practitioners in an *ad hoc* way, utilising different choices of p. We formalise the model slightly by calling it a first-degree fractional polynomial or FP1 function where the powers p are chosen from a restricted set, S (Royston and Altman, 1994). For pragmatic reasons, Royston and Altman (1994) suggested $S = \{-2, -1, -0.5, 0, 0.5, 1, 2, 3\}$, where Z^0 denotes $\log(Z)$. No subsequent changes to S have been proposed since then. The power transformation set includes the 'identiy' transformation $(p = 1)$, as well as the reciprocal, logarithmic, square root and square transformations. To fit an FP1 model each of the eight values of p is tried. The best-fitting model is defined as the one with the maximum likelihood among these models.

Extension from one-term FP1 functions to the more complex and flexible two-term FP2 functions follows immediately. FP2 functions with powers (p_1, p_2) include the formulations $\beta_1 Z^{p_1} + \beta_2 Z^{p_2}$ and $\beta_1 Z^{p_1} + \beta_2 Z^{p_1} \log(Z)$ if $p_2 = p_1$, the latter being the so-called repeated-powers functions. The best fit among the 36 (28 with $p_2 \neq p_1$ and 8 with $p_2 = p_1$) combinations of powers from S is defined as that with the highest likelihood.

An FPm model of degree m, $m = 1, 2$, is considered to have $2m$ degrees of freedom (d.f.), 1 d.f. for each parameter and 1 d.f. for each power. Because