Portable Parallel Seeds

Paul E. Johnson <pauljohn @ ku.edu>

October 30, 2012

Abstract

The R package portableParallelSeeds implements one method of managing random streams for batches of simulations. It is designed so that separate runs can be replicated exactly, in the sense proposed by Chambers (2008). Using the "many separate substreams" made possible by the CMRG random generator (as suggested by L'Ecuyer et al., 2002), we first create a large collection of initial states for many random streams, and then make those streams available to simulation runs. The random number streams for each separate run of a simulation are thus properly initialized. The framework allows one to run a simulation in a single workstation (iteratively) or on a cluster computer (parallel) and obtain the same results. It is also possible to select particular runs from a batch and re-start them for closer inspection. This approach allows for each separate run to depend on several separate streams of random numbers and it offers a method for changing among the random streams.

The portableParallelSeeds package for R proposes a simple, yet powerful, method for replicating simulations in a way that is valid across hardware types and operating systems. It is intended to facilitate the work of researchers who need to run a series of simulations, either on a desktop workstation or in a cluster of many separate computers. The approach proposed here allows the precise replication of the whole batch of runs, whether run in serial or parallel, but it has two special features that are not easily available elsewhere. First, any particular run of the model may be re-created, in isolation from the rest of the runs. Second, each particular run can be initialized with several separate streams of random numbers, thus making some simulation designs easier to implement. One can, for example, draw on two separate streams to initialize data for 1000 students and 50 teachers, and then draw random values from a third stream, and then turn back to the first stream to draw data for 50 more students from the same generator that generated the first batch of students. Thus, data for the same 1050 students would be obtained, whether they are drawn in two sets (1000+50) or in one set (1050).

The approach blends ideas about seed management from Chambers (2008) (as implemented in the R package SoDA, Chambers 2012) with ideas from the R package snowFT package by Hana Sevcikova and Tony R. Rossini (2010). Chambers proposes a method of recording the random generator's state that works well in simulations that run on a single piece of hardware, but it does not generalize directly to a cluster computing framework in which simulation runs begin separately on many separate nodes. The framework introduced in snowFT initializes each compute node with its own random seed, but does not separately initialize each run of the model. The plan used in portableParallelSeeds addresses these shortcomings.

1 Introduction

In statistical research, it is now common propose an estimator and then apply it to 1000s of simulated data sets in order to ascertain the sampling distribution (for a review, see Johnson, Forthcoming). Researchers face a variety of practical challenges in the management of these simulations so that the sources of variations in results across runs can be meaningfully understood. The ability to replicate runs within this research process is, quite obviously, of the first importance (Chambers, 2008).

1.1 Sketch of the portableParallelSeeds approach

Step 1. Create a Collection of Initializing States for Random Number Generators.

run	stream 1	stream 2	stream 3
1	1, 1	1,2	1,3
2	2,1	2,2	2,3
⋮			
2000	2000,1	2000,2	2000,3

Table 1: Matrix of Initializing States

It is necessary to conceptualize a simulation project as a sequence of separate "runs." A **run** is an isolated series of calculations that begins in a pre-determined state. Think of the collection of initializing states as a matrix that has one row for each anticipated run of the model. Within each row of this matrix, there will be information to initialize one or more separate streams of random numbers. The matrix of initializing states can be saved on disk. In that way, any particular run from a batch can be re-started on a different computer and the same results can be obtained. A sketch of this initializing matrix is offered in Table 1.

The creation of this set of initial random stream states is handled by the function `seedCreator()`. That function will create seeds for `nReps` separate runs, with `streamsPerRep` separate random stream initializers for each run. The S3 class of that object is `portableSeeds`.

Step 2. Design the simulation so that, when a run begins, the function that governs the run will retrieve its initializing states. Those states are then set into the R global environment. The portableParallelSeeds package provides function, `useStream`, to select among the random number streams.

The user's function should accept the run number and the object of type portableSeeds as arguments. For example,

```
> myFunction <- function(run, streamSet, a, b, c, d){
+     initPortableStreams(projSeeds = streamSet, run = run, verbose = FALSE)
+     ## simulation calculations based on parameters a, b, c, and d
+ }
```

If the initializing states are saved in a file, then re-starting the process on any computer running R will re-generate the same simulation because the R Core Team (2012) has taken great effort to assure us that saved R files can be transferred from one type of hardware to another.

1.2 Benefits of this Approach

The run-level initialization of random streams proposed here has several benefits.

Benefit 1. Get the same results for each individual run, every time, whether the exercise is conducted on a workstation (in a serial process) or on compute clusters of various sizes.

The approach in snowFT will initialize the compute nodes, but then repeatedly assign jobs to the nodes without re-setting the random streams. This will assure that a whole batch of simulations can be replicated on that particular hardware setup, but it does not assure replication on clusters of different sizes. If we have a cluster with 5 machines, each can be predictably initialized, and then the 1000 simulation runs will be assigned among the 5 nodes, one after the other. We cannot obtain the same result in a cluster with 10 nodes, however. We initialize 10 machines and the 1000 runs are assigned among them. The random streams assigned for runs that are assigned to machines 6 through 10 will be unique, so comparison of the runs against the first batch is impossible. The random streams used, for example, on the 6th run, will differ.

Benefit 2. Get the same results, even when a load balancing assignment of runs is used.

A load balancing algorithm monitors the compute nodes and sends the next assignment to the first available compute node. This may accelerate computations, but it plays havoc with replication. If 1000 runs are to be divided among 10 nodes, and are assigned in order to the same nodes, then the random number streams set on each node will remain in sequence across all of the runs. However, if we use a load balancing algorithm, then the jobs are not necessarily assigned to the same nodes on repeated runs. The presence of other programs running in a compute node or network traffic might slow down the completion of calculations, causing the node to "miss its spot in line," thus altering the assignment of all future runs among nodes. As a result, when a load balancing algorithm is used, replication of results for any particular run appears to be extremely unlikely, even if we always have access to the same number of nodes.

Benefit 3. Isolate runs and investigate them in detail.

In the process of exploring a model, it may be that some simulation runs are problematic. The researcher wants to know what's wrong, which usually involves re-starting the simulation and then exploring it interactively. Because each separate run begins with a set of saved random generator states, accurate replication is possible.

Because the approach proposed here allows each simulation to depend on several separate streams of randomness, the researcher has much more flexibility in conducting this investigation. For example, the "replication part" of the simulation might be restricted to draw from stream 1, while the researcher can change to stream 2 to draw more random values without changing the values that will be offered by stream 1 in the remainder of the simulation. This prevents gratuitous changes in simulated values from triggering sequence of unpredictable changes in simulation results.

Benefit 4. Isolate sources of randomness.

The approach proposed here can create several random streams for use within each run. Furthermore, each of these streams can be re-initialized at any point in the simulation run. This capability will help to address some problems that arise in applied research projects. It often happens that projects will ask a question about the effect of changing the sample size, for example, and they will draw completely fresh samples of size N and N+k, whereas they ought to draw exactly the same sample for the first N observations, and then draw k fresh observations after that. Otherwise, the effect of adding the k additional observations is confounded with the entire replacement of the original N observations. Because portableParallelSeeds offers several separate streams, and each can be re-set at any time, the correct implementation is more likely to be achieved.

There are other scenarios in which the separate streams may be valuable. A project designer might conceptualize a single run as a family of small variations on a theme. Within each re-start in the family, several variables need to be replicated exactly, while others must be new. Because several streams are available, this can be managed easily.

The several separate streams are not absolutely necessary, but they will make it easier to isolate sources of change in a project. Drawing a single number from a shared random generator will put all of the following draws "out of sequence" and make replication of succeeding calculations impossible. It makes sense to segregate those calculations so that they draw from a separate random generator stream.

2 A Brief Soliloquy on Random Generators

While working on this project, I've gained some insights about terminology and usage of random generators. A brief review may help readers follow along with the presentation.

2.1 Terminology

A **pseudo random number generator** (PRNG) is an object that offers a stream of numbers. We treat those values as though they are random, even though the PRNG uses deterministic algorithms to generate them (that is why it is a "pseudo" random generator). From the perspective of the outside observer who is not privy to the details about the initialization of the PRNG, each value in the stream of numbers appears to be an equally likely selection among the possible values. [1]

Inside the PRNG, there is a vector of values, the **internal state** of the generator, which is updated as values are drawn. The many competing PRNG designs generally have unique (and not interchangeable) internal state vectors. Sometimes that internal state is called the **seed** of the PRNG because it represents the current position from which the next value is to be drawn. The term seed is also used with a different meaning by applied researchers. For them, a seed is an integer that starts up a generator in a given state. This other usage is, strictly speaking, technically incorrect, but it is used widely. For example, the SAS Language Reference states, "Random-number functions and CALL routines generate streams of pseudo-random numbers from an initial starting point, called a seed, that either the user or the computer clock supplies" (2011). It would be more correct to say the user supplies an "intializing integer." In any context where confusion is possible, I'll refer to these values as the internal state vector and the initializing integer.

The values from the generator are used as input in procedures that simulate draws from **statistical distributions**, such as the uniform or normal distributions. The conversion from the generator's output into random draws from distributions is a large field of study, some distributions are very difficult to approximate. The uniform distribution is the only truly easy distribution. If the PRNG generates integer values, we simply divide each random integer by the largest possible value of the PRNG to obtain equally likely draws from the $[0, 1]$ interval. It is only slightly more difficult to simulate draws from some distributions (e.g, the logistic), while for others (e.g., the gamma distribution) simulation are considerably more difficult. The normal distribution, which occupies such a central place in statistical theory, is an in-between case for which there are several competing proposals.

2.2 Differentiate "seed" from "internal state".

The confusion between applied researchers who think of the seed as an initial state, and software developers who think of the seed as an internal state that evolves, comes to the forefront in the discussion of replication.

Most R users have encountered the set.seed() function. We run, for example,

```
> set.seed(12345)
```

The argument "12345" is not a "seed". It is an integer that is used to re-set the generator's internal state to a known position. It is important to understand that the internal state of the random generator is not "12345". The generator's internal state, the vector of numbers that the generator uses over time, does not begin at a value "12345". Instead, the internal state is a much more elaborate thing. Consider just the first 10 elements of the generator's internal state (the thing the experts call the seed) in R (by viewing the variable `.Random.seed`):

```
> s0 <- .Random.seed
> s0[1:10]
```

```
 [1]         403         624  1638542565   108172386 -1884566405 -1838154368
 [7]  -250773631   919185230 -1001918601 -1002779316
```

I'm only displaying the first 10 values (out of 626) of the initial state of the default random generator, which is the Mersenne-Twister (hereafter referred to as MT19937). The Mersenne-Twister was proposed by **?** and, at the current time, it is considered the premier random generator for simulations conducted on workstations. It is now the default random generator in almost every program used for statistical research (including R, SAS, Matlab, Mplus, among others).

How can you check my claim that the default generator is MT19937? Run

```
> RNGkind()
```

```
[1] "Mersenne-Twister" "Inversion"
```

The output includes two values, the first is the name of the existing random generator. The second value is the algorithm that is used to simulate values from a normal distribution.

2.3 MT19937's internal state

In order to understand the way R implements the various PRNGs, and thus the way portableParallelSeeds works, it is important to explore what happens to the internal state of the generator as we draw random numbers. Since we began with the default, MT19937, we might as well work on that first. Suppose we draw one value from a uniform distribution.

```
> runif(1)
```

```
[1] 0.7209039
```

Take a quick look at the generator's internal state after that.

```
> s1 <- .Random.seed
> s1[1:10]
```

```
 [1]         403           1 -1346850345   656028621    13211492  1949688650
 [7]    95765173 -1737862641   -58526954  1501289920
```

The interesting part is in the first two values.

- 403. This is a value that R uses to indicate which type of generator created this particular state vector. The value "03" indicates that MT19937 is in use, while the value "4" means that the inversion method is used to simulate draws from a normal distribution. The Mersenne-Twister is the default random generator in R (and most good programs, actually).
- 1. That's a counter. How many random values have been drawn from this particular vector? Only one.

Each time we draw another uniform random value, the generator's counter variable will be incremented by one.

```
> runif(1)
```

```
[1] 0.8757732
```

```
> s2 <- .Random.seed
> runif(1)
```

```
[1] 0.7609823
```

```
> s3 <- .Random.seed
> runif(1)
```

```
[1] 0.8861246
```

```
> s4 <- .Random.seed
> cbind(s1, s2, s3, s4)[1:8, ]
```

```
              s1          s2          s3          s4
[1,]         403         403         403         403
[2,]           1           2           3           4
[3,] -1346850345 -1346850345 -1346850345 -1346850345
[4,]   656028621   656028621   656028621   656028621
[5,]    13211492    13211492    13211492    13211492
[6,]  1949688650  1949688650  1949688650  1949688650
[7,]    95765173    95765173    95765173    95765173
[8,] -1737862641 -1737862641 -1737862641 -1737862641
```

I'm only showing the first 8 elements, to save space, but there's nothing especially interesting about elements 9 through 626. They are all are integers, part of a complicated scheme that **?** created. The important point is that integers 3 through 626 are exactly the same in s1, s2, s3, and s4. They will stay the same until we draw 620 more random numbers from the stream.

As soon as we draw more random numbers–enough to cause the 2nd variable to increment past 624–then the *whole vector* changes. I'll draw 620 more values. The internal state s5 is "on the brink" and one more random uniform value pushes it over the edge. The internal state s6 represents a wholesale update of the generator.

```
> invisible(runif(620))
> s5 <- .Random.seed
> invisible(runif(1))
> s6 <- .Random.seed
> invisible(runif(1))
> s7 <- .Random.seed
> invisible(runif(1))
> s8 <- .Random.seed
> cbind(s1, s5, s6, s7, s8)[1:8, ]
```

```
              s1           s5           s6           s7           s8
[1,]         403          403          403          403          403
[2,]           1          624            1            2            3
[3,] -1346850345  -1346850345   1750213233   1750213233   1750213233
[4,]   656028621    656028621   1893020862   1893020862   1893020862
[5,]    13211492     13211492   1799303033   1799303033   1799303033
[6,]  1949688650   1949688650   1075042007   1075042007   1075042007
[7,]    95765173     95765173   1631350616   1631350616   1631350616
[8,] -1737862641  -1737862641    746260959    746260959    746260959
```

After the wholesale change between s5 and s6, another draw produces more "business as usual." Observe that the internal state of the generator in columns s6, s7 and s8 is not changing, except for the counter.

Like all R generators, the MT19937 generator can be re-set to a previous saved state. There are two ways to do this. One way is the somewhat restrictive function `set.seed()`. That translates an initializing integer into the 626 valued internal state vector of the generator (that's stored in `.Random.seed`).

```
> set.seed(12345)
> runif(1)
```

```
[1] 0.7209039
```

```
> s9 <- .Random.seed
```

We can achieve the same effect by using the assign function to replace the current value of .Random.seed with a copy of a previously saved state, s0. I'll draw one uniform value and then inspect the internal state of the generator (compare s1, s9, and s10).

```
> assign(".Random.seed", s0, envir=.GlobalEnv)
> runif(1)
```

```
[1] 0.7209039
```

```
> s10 <- .Random.seed
> cbind(s1, s9, s10)[1:8, ]
```

```
              s1           s9          s10
[1,]         403          403          403
[2,]           1            1            1
[3,] -1346850345  -1346850345  -1346850345
[4,]   656028621    656028621    656028621
[5,]    13211492     13211492     13211492
[6,]  1949688650   1949688650   1949688650
[7,]    95765173     95765173     95765173
[8,] -1737862641  -1737862641  -1737862641
```

The reader should notice that after re-initializing the state of the random generator, we draw the exact same value from `runif(1)` and after that the state of the generator is the same in all of the cases being compared (s9 is the same as s10).

The MT19937 is a great generator with a very long repeat cycle. The cycle of values it provides will not begin to repeat itself until it generates 2^{19937} values. It performs very well in a series of tests of random number streams.

The only major shortcoming of MT19937 is that it does not work well in parallel programming. MT19937 can readily provide random numbers for 1000s of runs of a simulation on a single workstation, but it is very difficult to initialize MT19937 on many compute nodes in a cluster so that the random streams are not overlapping. One idea is to spawn separate MT19937 generators with slightly different internal parameters so that the streams they generate will differ (Mascagni, Ceperley and Srinivasan, 2000; see also Matsumoto and Nishimura, 2000). For a variety of reasons, work on parallel computing with an emphasis on replication has tended to use a different PRNG, which is described next.

2.4 CMRG, an alternative generator.

In parallel computing with R, the most widely used random generator is Pierre L'Ecuyer's combined multiple-recursive generator, or CMRG L'Ecuyer (1999).

R offers a number of pseudo random generators, but only one random generator can be active at a given moment. That restriction applies because the variable .Random.seed is used as the central co-ordinating piece of information. When the user asks for a uniform random number, the R internal system scans the .Random.seed to find out which PRNG algorithm should be used and then the value of .Random.seed is referred to the proper generator.

We ask R to use that generator by this command:

```
> RNGkind("L'Ecuyer-CMRG")
```

That puts the value of .Random.seed to a proper condition in the global environment. Any R function that depends on random numbers–to simulate random distributions or to initialize estimators–it will now draw from the CMRG using .Random.seed as its internal state.

Parallel computing in a cluster of separate systems pre-supposes the ability to draw separate, uncorrelated, non-overlapping random numbers on each system. In order to do that, we follow an approach that can be referred to as the "many separate substreams" approach. The theory for this approach is elegant. Think of a really long vector of randomly generated integers. This vector is so long it is, well, practically infinite. It has more numbers than we would need for thousands of separate projects. If we divide this practically infinite vector into smaller pieces, then each piece can be treated as its own random number stream. Because these separate vectors are drawn from the one really long vector of random numbers, then we have confidence that the separate substreams are not overlapping each other and are not correlated with each other. But we don't want to run a generator for a really long time so that we can find the subsections of the stream. That would require an impractically huge amount of storage. So, to implement the very simple, solid theory, we just need a practical way to splice into a random vector, to find the initial states of each separate substream.

That sounds impossible, but a famous paper by (L'Ecuyer et al., 2002) showed that it can be done. L'Ecuyer et al. demonstraed an algorithm that can "skip" to widely separated points in the long sequence of random draws. Most importantly, this is done *without actually generating the practically infinite series of values*. In R version 2.14, the L'Ecuyer CMRG was included as one of the available generators, and thus it became possible to implement this approach. We can find the generator's internal state at far-apart positions.

Lets explore L'Ecuyer's CMRG generator, just as we explored MT19937. First, we tell R to change its default generator, and then we set the initial state and draw four values. We collect the internal state (.Random.seed) of the generator after each random uniform value is generated.

```
> RNGkind("L'Ecuyer-CMRG")
> set.seed(12345)
> t0 <- .Random.seed
> runif(1)
```

```
[1] 0.0724409
```

```
> t1 <- .Random.seed
> runif(1)
```

```
[1] 0.7698878
```

```
> t2 <- .Random.seed
> runif(1)
```

```
[1] 0.3254684
```

```
> t3 <- .Random.seed
> rnorm(1)
```

```
[1] 0.9883728
```

```
> t4 <- .Random.seed
> cbind(t1, t2, t3, t4)
```

```
              t1          t2          t3          t4
[1,]          407         407         407         407
[2,]   1638542565   108172386   684087654 -1951841990
[3,]    108172386   684087654  1019552775  1064477516
[4,]    684087654  1019552775 -1951841990  -537073593
[5,]  -1838154368  -250773631   372956394   945249426
[6,]   -250773631   372956394  2007876921  1758050460
[7,]    372956394  2007876921   945249426   998522591
```

Apparently, this generator's assigned number inside the R framework is "07" (the "4" still indicates that inversion is being used to simulate normal values). There are 6 integer numbers that characterize the state of the random generator. The state vector is thought of as 2 vectors of 3 elements each. Note that the state of the CMRG process does not include a counter variable comparable to the 2nd element in the MT19937's internal state. Each successive draw shifts the values in those vectors.

The procedure to skip ahead to the starting point of the next substream is implemented in the R function nextRNGStream, which is provided in R's parallel package. The state vectors, which can be used to re-initialize 5 separate random streams, are shown below.

```
> require(parallel) ## for nextRNGStream
> substreams <- vector("list", 5)
> substreams[[1]] <- t0
> substreams[[2]] <- nextRNGStream(t0)
> substreams[[3]] <- nextRNGStream(substreams[[2]])
> substreams[[4]] <- nextRNGStream(substreams[[3]])
> substreams[[5]] <- nextRNGStream(substreams[[4]])
> substreams
```

```
[[1]]
[1]         407 -2132566924  1638542565   108172386 -1884566405 -1838154368
[7]  -250773631

[[2]]
[1]         407 -1645818963   548746318   440099794   143370804  -548492161
[7]   249546247

[[3]]
[1]        407  -363950260  -864007039 1529726914  -409305868  -670976700   723026206

[[4]]
[1]        407  -310576051  -443186231 1234569748    76923062 1387306546  -309616276

[[5]]
[1]         407 -1515242020 -1576741206   -11449651  -783708047  1716218842
[7]   627172014
```

2.5 rnorm draws two random values, but runif draws only one. rgamma is less predictable!

One important tidbit to remember is that simulating draws from some distributions will draw more than one number from the random generator. This disturbs the stream of values coming from the random generator, which causes simulation results to diverge.

Here is a small example in which this problem might arise. We draw 3 collections of random numbers.

```
> set.seed(12345)
> x1 <- runif(10)
> x2 <- rpois(10, lambda=7)
> x3 <- runif(10)
```

Now suppose we decide to change the variable x2 to draw from a normal distribution.

```
> set.seed(12345)
> y1 <- runif(10)
> y2 <- rnorm(10)
> y3 <- runif(10)
> identical(x1,y1)
```

```
[1] TRUE
```

```
> identical(x2,y2)
```

```
[1] FALSE
```

```
> identical(x3, y3)
```

```
[1] FALSE
```

```
> rbind(x3, y3)
```

```
        [,1]      [,2]       [,3]       [,4]      [,5]      [,6]      [,7]
x3 0.3390028 0.8422707 0.50308216 0.02741534 0.8661977 0.4883648 0.1443217
y3 0.1042063 0.9140845 0.09050534 0.14816555 0.9519876 0.9792253 0.1993882
        [,8]      [,9]     [,10]
x3 0.8255914 0.6919255 0.5762445
y3 0.5473667 0.6811563 0.2191464
```

In these two cases, we draw 30 random numbers. I expect that x1 and y1 will be identical, and they are. I know x2 and y2 will differ. But I expected, falsely, that x3 and y3 would be the same. But they are not. Their values are not even remotely similar. If we then go to to make calculations and compare these two models, then our conclusions about the effect of changing the second variable from poisson to normal would almost certainly be incorrect, since we have accidentally caused a wholesale change in y3 as well.

Why does this particular problem arise? The function rnorm() draws two values from the random generator, thus causing all of the uniform values in y3 to differ from x3. This is easiest to see with MT19937, since that generator offers us the counter variable in element 2. I will re-initialize the stream, and then draw some values.

```
> RNGkind("Mersenne-Twister")
> set.seed(12345)
> runif(1); s1 <- .Random.seed
```

```
[1] 0.7209039
```

```
> runif(1); s2 <- .Random.seed
```

```
[1] 0.8757732
```

```
> runif(1); s3 <- .Random.seed
```

```
[1] 0.7609823
```

```
> rnorm(1); s4 <- .Random.seed
```

```
[1] 1.206173
```

```
> cbind(s1, s2, s3, s4)[1:8, ]
```

```
             s1          s2          s3          s4
[1,]        403         403         403         403
[2,]          1           2           3           5
[3,] -1346850345 -1346850345 -1346850345 -1346850345
[4,]  656028621   656028621   656028621   656028621
[5,]   13211492    13211492    13211492    13211492
[6,] 1949688650  1949688650  1949688650  1949688650
[7,]   95765173    95765173    95765173    95765173
[8,] -1737862641 -1737862641 -1737862641 -1737862641
```

Note that the counter jumps by two between s3 and s4.

The internal counter in MT19937 makes the "normal draws two" problem easy to spot. With CMRG, this problem is more difficult to diagnose. Since we know what to look for, however, we can replicate the problem with CMRG. We force the generator back to the initial state and then draw five uniform random variables.

```
> assign(".Random.seed", t1, envir=.GlobalEnv)
> u1 <- .Random.seed
> invisible(runif(1))
> u2 <- .Random.seed
> invisible(runif(1))
> u3 <- .Random.seed
> invisible(runif(1))
> u4 <- .Random.seed
> invisible(runif(1))
> u5 <- .Random.seed
> cbind(u1, u2, u3, u4, u5)
```

```
              u1          u2          u3          u4          u5
[1,]         407         407         407         407         407
[2,]  1638542565   108172386   684087654  1019552775 -1951841990
[3,]   108172386   684087654  1019552775 -1951841990  1064477516
[4,]   684087654  1019552775 -1951841990  1064477516  -537073593
[5,] -1838154368  -250773631   372956394  2007876921   945249426
[6,]  -250773631   372956394  2007876921   945249426  1758050460
[7,]   372956394  2007876921   945249426  1758050460   998522591
```

The internal states are displayed. Note the state of the generator u5 is the same as t4 in the previous section, meaning that drawing 5 uniform random variables puts the CMRG into the same state that CMRG reaches when we draw 3 uniform values and 1 normal variable.

The situation becomes more confusing when random variables are generated by an accept/reject algorithm. If we draw several values from a gamma distributions, we note that MT19937's counter may change by 2, 3, or more steps.

```
> RNGkind("Mersenne-Twister")
> set.seed(12345)
> invisible(rgamma(1, shape = 1)); v1 <- .Random.seed[1:4]
> invisible(rgamma(1, shape = 1)); v2 <- .Random.seed[1:4]
> invisible(rgamma(1, shape = 1)); v3 <- .Random.seed[1:4]
> invisible(rgamma(1, shape = 1)); v4 <- .Random.seed[1:4]
> invisible(rgamma(1, shape = 1)); v5 <- .Random.seed[1:4]
> invisible(rgamma(1, shape = 1)); v6 <- .Random.seed[1:4]
> cbind(v1, v2, v3, v4, v5, v6)
```

```
              v1          v2          v3          v4          v5          v6
[1,]         403         403         403         403         403         403
[2,]           2           4           7           9          11          16
[3,] -1346850345 -1346850345 -1346850345 -1346850345 -1346850345 -1346850345
[4,]   656028621   656028621   656028621   656028621   656028621   656028621
```

Most of the time, drawing a single gamma value uses just 2 or 3 numbers from the generator, but about 10 percent of the time more draws will be taken from the generator. [2]

The main point in this section is that apparently harmless changes in the design of a program may disturb the random number stream, thus making it impossible to replicate the calculations that follow the disturbance. Anticipating this problem, it can be essential to have access to several separate streams within a given run in order to protect against accidents like this.

Many other functions in R may draw random values from the stream, thus throwing off the sequence that we might be depending on for replication. Many sorting algorithms draw random numbers, thus altering the stream for successive random number generation. While debugging a program, one might unwittlingly insert functions that exacerbate the problem of replicating draws from random distributions. If one is to be extra-careful on the replication of random number streams, it seems wise to keep a spare stream for every project and then switch the generator to use that spare stream, and then change back to the other streams when number that need to be replicated are drawn.

3 Example Usage of portableParallelSeeds

The following uses seedCreator to generate initializing states for 1000 simulation runs. In each of them we allow for three streams. The collection of random generator states is returned as an R object projSeeds, but it is also written on disk in a file called “fruits.rds”.

```
> library(portableParallelSeeds)
> projSeeds <- seedCreator(1000, 3, seed = 123456, file = "fruits.rds")
> A1 <- projSeeds[[787]]
> A1 ## shows states of 3 generators for run 787
```

```
[[1]]
[1]        407  -491020330   555536868  2085569258 -2036950451   895819634
[7]  180773870

[[2]]
[1]        407  1300088217 -1122483900  -780413849 -2028680486   876870054
[7] 1794711846

[[3]]
[1]        407 -1581640375  -278790076   -24170581   537304202  -881783055
[7] -886305875
```

For no particular reason, I elected to explore the seeds saved for run 787 in this example. We first check that the initPortableStreams() function can receive the collection of initializing information and re-generate the streams for run 787.

```
> initPortableStreams(projSeeds, run = 787, verbose = TRUE)
```

```
[1] "initPortableStreams, Run =  787"
[1]        407  -491020330   555536868  2085569258 -2036950451   895819634
[7]  180773870
[1] "CurrentStream CurrentStream CurrentStream = 1"
[1] "All Current States"
[1] "c(407, -491020330, 555536868, 2085569258, -2036950451, 895819634, 180773870)"
[2] "c(407, 1300088217, -1122483900, -780413849, -2028680486, 876870054, 1794711846)"
[3] "c(407, -1581640375, -278790076, -24170581, 537304202, -881783055, -886305875)"
```

```
> .Random.seed
```

```
[1]        407  -491020330   555536868  2085569258 -2036950451   895819634
[7]  180773870
```

```
> getCurrentStream()
```

```
[1] 1
```

```
> runif(4)
```

```
[1] 0.58072178 0.74450456 0.49674707 0.06439554
```

Next, verify that if we read the file "fruits.rds", we can obtain the exact same set of initializing states for run 787. Note that the 4 random uniform values that are drawn exactly match the 4 values drawn in the previous code section.

```
> myFruitySeeds <- readRDS("fruits.rds")
> B1 <- myFruitySeeds[[787]]
> identical(A1, B1) # check
```

```
[1] TRUE
```

```
> initPortableStreams("fruits.rds", run=787)
> .Random.seed
```

```
[1]        407  -491020330   555536868  2085569258 -2036950451   895819634
[7]  180773870
```

```
> runif(4)
```

```
[1] 0.58072178 0.74450456 0.49674707 0.06439554
```

That should be sufficient to satisfy our curiosity, a more elaborate test is offered next.

Step 2 in the simulation process is the creation of a function to conduct one single run of the simulation exercise. This function draws N normal variables from stream 1, some poisson variates from stream 2, then returns to stream 1 to draw another normal observation. We will want to be sure that the N+1 normal values that are drawn in this exercise are the exact same normals that we would draw if we took N+1 values consecutively from stream 1.

```
> runOneSimulation <- function(run, streamsource, N, m, sd){
+ initPortableStreams(streamsource, run = run, verbose= FALSE)
+ datX <- rnorm(N, mean = m, sd = sd)
+ datXmean <- mean(datX)
+ useStream(2)
+ datY <- rpois(N, lambda = m)
+ datYmean <- mean(datY)
+ useStream(1)
+ datXplusOne <- rnorm(1, mean = m, sd = sd)
+ ## Should be N+1'th element from first stream
+ c("datXmean" = datXmean, "datYmean" = datYmean, "datXplusOne" = datXplusOne)
+ }
```

Now we test the framework in various ways, running 1000 simulations with each approach. Note the objects serial1, serial2, and serial3 are identical.

```
> ## Give seed collection object to each simulation, let each pick desired seed
> serial1 <- lapply(1:1000, runOneSimulation, projSeeds, N=800, m = 14, sd = 10.1)
> ## Re-load the seed object, then give to simulations
> fruits2 <- readRDS("fruits.rds")
> serial2 <- lapply(1:1000, runOneSimulation, fruits2, N=800, m = 14, sd = 10.1)
> ## Re-load file separately in each run (is slower)
> serial3 <- lapply(1:1000, runOneSimulation, "fruits.rds", N = 800, m = 14, sd=10.1)
> identical(serial1, serial2)
```

```
[1] TRUE
```

```
> identical(serial1, serial3)
```

```
[1] TRUE
```

Now, lets check the N+1 random normal values. We need to be sure that the 801'th random normal from stream 1 is equal to the 3'rd element in the returned vector. Lets check run 912. First, re-initialize the run, and then draw 801 new values from the normal generator (with the default stream 1).

```
> initPortableStreams("fruits.rds", run = 912, verbose = FALSE)
> .Random.seed
```

```
[1]         407  -947703512 -1521217876   961167881 -1727596281  1232391401
[7] -1268975159
```

```
> X801 <- rnorm(801, m=14, sd = 10.1)
> X801[801]
```

```
[1] 16.93997
```

'The value displayed in variable X801[801] should be identical to the third element in the returned value that was saved in the batch of simulations. Observe

```
> serial1[[912]]
```

```
   datXmean    datYmean datXplusOne
   14.10043    14.16125    16.93997
```

Bingo. The numbers match. We can draw understandably replicatable streams of random numbers, whether we draw 800, switch to a different stream, and then change back to draw another, and obtain the same result if we just draw 801 in one block.

```
> unlink("fruits.rds") #delete file
```

4 Frequently Asked Questions

5 Conclusion

This paper describes an R package called portableParallelSeeds. The package provides functions that can generate a seed collection which can be put to use in a series of simulation runs. The approach described here is based on the re-initialization of individual runs, and thus it will work whether the simulations are conducted on a single computer or in a cluster computer.

The tools provided are intended to help with situations like the following.

Problem 1. I scripted up 1000 R runs and need high quality, unique, replicable random streams for each one. Each simulation runs separately, but I need to be confident their streams are not correlated or overlapping. I need to feel confident that the results will be the same, whether or not I run these in a cluster with 4 computers, or 4000 computers.

Problem 2. For replication, I need to be able to select any run, and restart it exactly as it was. That means I need pretty good record keeping along with a simple approach for bringing simulations back to life.

Question: Why is this better than the simple old approach of setting the generator's initial state within each run with a formula like this.

set.seed(2345 + 10 * run)

Answer: That does allow replication, but it does not assure that each run uses non-overlapping random number streams. It offers absolutely no assurance whatsoever that the runs are actually non-redundant.

Nevertheless, it is a method that is widely used and recommended by some visible HOWTO guides.

Notes

[1]Someone who observes thousands of values from the PRNG may be able to deduce its parameters and reproduce the stream. If we are concerned about that problem, we can add an additional layer of randomization that shuffles the output of the generator before revealing it to the user.

[2]A routine generates 10,000 gamma values while tracking the number of values drawn from the random generator for each is included with portableParallelSeeds in the examples folder (gamma_draws.R).

References

Chambers, John M. 2008. *Software for data analysis: programming with R.* Statistics and computing New York ; London: Springer. (document)

Chambers, John M. 2012. "SoDA: Functions and Exampels for "Software for Data Analysis".". R package version 1.0-4.
URL: *http://CRAN.R-project.org/package=SoDA* (document)

L'Ecuyer, Pierre. 1999. "Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators." *Operations Research* 47(1):159–164. 2.4

L'Ecuyer, Pierre, Richard Simard, E. Jack Chen and W. David Kelton. 2002. "An Object-Oriented Random-Number Package with Many Long Streams and Substreams." *Operations Research* 50(6):1073–1075. (document), 2.4

Mascagni, Michael, David Ceperley and Ashok Srinivasan. 2000. "SPRNG: A Scalable Library for Pseudorandom Number Generation." *ACM Transactions on Mathematical Software* 26:436—461. 2.3

Matsumoto, M and T. Nishimura. 2000. Dynamic Creation of Pseudorandom Number Generators. In *Monte Carlo and Quasi-Monte Carlo methods*, ed. H. Niederreiter and J. Spanier. Springer pp. 56–69. 2.3

SAS Institute. 2011. *SAS 9.2 Language Reference: Dictionary.* Fourth edition ed. Cary, NC: SAS Institute. Using Random-Number Functions and CALL Routines.
URL: *http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a001281561.htm* 2.1

Sevcikova, Hana and A. J. Rossini. 2010. *snowFT: Fault Tolerant Simple Network of Workstations.* R package version 1.2-0.
URL: *http://CRAN.R-project.org/package=snowFT* (document)