Portable Parallel Seeds

Paul E. Johnson <pauljohn @ ku.edu>

November 2, 2015

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 proposed 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 facilitates 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 phases or in one continuous series of random draws.

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 (Sevcikova and Rossini 2012). 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 to 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). One of the practical problems is to manage the insertion of randomness into the analysis.

Researchers face a variety of practical challenges in the management of these simulations. It is vital that each simulation exercise be replicatable, even in its very fine-grained details (Chambers, 2008). That is to say, in order to feel confident in the results of a simulation exercise, we need to be able to re-generate its runs exactly, not just on average. Suppose we conduct 2000 repetitions (runs) in a batch of simulations. We should be able to repeat the whole batch, exactly, and we should also be able to repeat individual part of the exercise. The simulated random values drawn for the 1324rd line of a data structure should be the same whether we re-start a project on a cluster computer or in a desktop workstation. If one accidentally collects the same random numbers for two variables that are expected to differ, one may mistakenly conclude that there is no difference to be found. On the other hand, if one accidentally draws numbers that differ, when the logic indicates that they ought to be exactly the same, then one will mistakenly conclude that something important has changed. The tools offered in portableParallelSeeds will help Monte Carlo studies avoid some of those dangers.

1.1 A Quick For Loop Example

Because the discussion might seem intimidating to newcomers (and overflowing with jargon), lets begin with the simple use case. A researcher wants to run a Monte Carlo calculation 1000 times. Because each of these simulations might have some abnormal result, it is important that one must be able to pick a particular run, for example 232, and re-initialize it exactly. The following creates a seed collection called projSeeds and then the for loop accesses these seeds, one after the other.

```
library(portableParallelSeeds)
projSeeds <- seedCreator(1000, 3, seed = 232323)
x1 <- vector(1000, mode = "numeric")
for(i in 1:1000) {
     setSeeds(projSeeds, run = i)
     x1[i] <- mean(rnorm(100))
 }
```

Now suppose the researcher notices that one simulation is somehow unusual. Perhaps a histogram reveals that one simulation generates a result which appears to be an outlier. Perhaps the researcher will be interested to find out more about that one case. For example, she finds the lowest average,

```
lowest <- which.max(x1); x1[lowest]
```

```
[1] 0.2757631
```

To more carefully inspect the lowest simulation, it is possible to ask for just that one run, and re-start it from an identical initial point.

```
setSeeds(projSeeds, run = lowest)
x1.lowest <- rnorm(100)
(x1.lowest.mean <- mean(x1.lowest))
```

```
[1] 0.2757631
```

Note that the calculated mean is identical to the value from the previous simulation. This is not a very interesting simulation, of course, but the main idea should be clear enough. One can step into the batch of models and re-start one run from a position that exactly replicates the initial trouble case.

Finally, suppose one wanted to re-run a whole batch of simulations from an identical spot. Perhaps we add more elaborate calculations, drawing additional observations from the random number stream. We need to be sure we subject this new, enhanced procedure to an identical sample.

```
x2 <- vector(1000, mode = "numeric")
for(i in 1:1000) {
     setSeeds(projSeeds, run = i, verbose = FALSE)
     x2[i] <- mean(rnorm(100))
 }
    identical(x1, x2)
```

```
[1] TRUE
```

We confirm, then, it is possible to re-initialize the random generator's stream so as to re-create the same results.

1.2 Runs, streams: the portableParallelSeeds approach

A **run** is an isolated series of calculations that begins in a pre-determined state. A random number **stream**, which is created by a pseudo-random number generator, is an identical sequence of values obtained by repeated calls on the generator. (A PRNG is not a "random" sequence in the colloquial sense. It is rather a sequence that is unpredictable to one who is not privy to the initializing state of the random stream.) Experimental runs differ because each begins in a different state. The draws from the generator are uncorrelated and do not have repetitive patterns.

The design of functions that use random numbers in R is a little bit tricky because all generators use to a single state variable called `.Random.seed` that is stored in the global environment of the R session. After each draw from the random generator, `.Random.seed` is updated inside the R global environment. As mentioned in Gentleman, "The decision to have these [random generator] functions manipulate a global variable, `.Random.seed`, is slightly unfortunate as it makes it somewhat more difficult to manage several different random number streams simultaneously" (Gentleman, 2008, p. 201).

The portableParallelSeeds package finesses a solution to that limitation in the following way. Information for several separate streams resides in memory. Random number draws continue to pull from a stream until the user asks R to start drawing from a different stream. Techincally, beneath the surface, we are swapping a different obect in place of R's `.Random.seed` when we intend to draw from a particular stream. This solution is not technically challenging, but it was somewhat difficult to package in a way that will be easy to use and trouble free for most users (see the function `useStream()` for information about how to select among the random number streams).

1.3 Runs, streams: the portableParallelSeeds approach

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

Run the `seedCreator()` function to create initializing information for a a given number of runs (and streams within each run). The object can reside in memory, but we generally recommend that users should specify the `file` argument so that a copy of the information is saved in a physical file. The streams can be stored in a file and transported among systems so that a simulation can be re-started on various operating systems.

It may be helpful to 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. 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 a function that isolates calculations for each run, and call it over and over.

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

Among the first actions within this function should be the establishment of the replicable random number streams. Basically, we simply take the initializing states from the seed warehouse and make them available in an R environment called `.pps`. The function used most commonly for that, `setSeeds()` is used as follows: `setSeeds(projSeeds, run = i)`. That will take an entire collection of seeds `projSeeds` into memory and then it will set the i'th one as the current set of streams. In an earlier version of this package, this was viewed as a two step process, one would select a row from the seed warehouse and then set it in place. That approach, called `setSeedCollection`, can still be used. The syntax would be `setSeedCollection(projSeeds[i])`.

A simulation program should have a function that orchestrates one "run" of calculations. If that function is called, for example, `conductOneRun()`, then the first piece of business in that function will be to re-set the set of random generator states. For example, given a run number and a collection of seeds, the work of initializing the simulation can be passed along to `setSeeds()`.

```
conductOneRun <- function(run, streamSet, a, b, c, d){
     setSeeds(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 (2014) has taken great effort to assure us that saved R files can be transferred from one type of hardware to another.

1.4 Benefits of this Approach

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

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

By comparison, this is not possible with snowFT(Sevcikova and Rossini, 2012). 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 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  180773870
```

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

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

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

```
setSeeds(projSeeds, run = 787, verbose = TRUE)
```

```
[1] "setSeeds, Run =  787"
[1]          407   -491020330    555536868   2085569258  -2036950451    895819634    180773870
[1] "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    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
```

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

```
[1]          407   -491020330    555536868   2085569258  -2036950451    895819634    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){
      setSeeds(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).

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

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

3 Frequently Asked Questions

4 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.

References

Chambers, John M. 2008. *Software for data analysis: programming with R.* Statistics and computing New York ; London: Springer. (document), 1

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)

Gentleman, Robert. 2008. *R Programming for Bioinformatics.* 1 edition ed. Boca Raton: Chapman and Hall/CRC. 1.2

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)

Sevcikova, Hana and A. J. Rossini. 2012. *snowFT: Fault Tolerant Simple Network of Workstations.* R package version 1.3-0.
URL: *http://CRAN.R-project.org/package=snowFT* (document), 1.4