Pseudo Random Generator Basics

Paul E. Johnson <pauljohn @ ku.edu>

August 6, 2014

Abstract

While working on the R package portableParallelSeeds, I realized that there are a lot of little details about pseudo random generators that are not well understood.

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.

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. The differences in the structure of the internal state is one of the most striking facts that we gather while looking under the hood of the random generator code. It is common to refer to that internal state as the **seed** of the PRNG. It represents the current position from which the next value is to be drawn.

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 The Difference between the "seed" and the "internal state".

The term "seed" is often used incorrectly, or at least differently, by applied researchers. For them, a seed is an integer that starts up a generator in a given state. This misunderstanding flows from mis-statements in the documentation for commonly used software packages. 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." From that initializing integer, the software does the work to create a unique initial internal state of the generator (a seed). To avoid re-creating more confusion, I will avoid the term seed wherever possible, instead referring to the internal state vector and the initializing integer.

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 initializing integer, which is used by the set.seed function 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, is actually a more complicated structure. The integer value "12345" is important only because it is used by the set.seed() function to construct that more complicated internal initial state.

There are several random generators supplied with R. The default is the Mersenne-Twister, commonly known as MT19937 Matsumoto and Nishimura (1998). Consider just the first 12 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[1:12]
```

```
 [1]          403          624   1638542565    108172386 -1884566405 -1838154368
 [7] -250773631  919185230 -1001918601 -1002779316  -321961507  1781331706
```

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

The generator that is currently selected for use in R can be revealed by this command,

```
> 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.

2.1 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.

2.2 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
```

3 Replication Challenges

3.1 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.

3.2 Be cautious about mvrnorm.

The MASS package function mvrnorm() is very widely used to generate multivariate normal data. It creates one row of data for each sample requested.

I recently noticed a quirk while trying to replicate some results. Suppose we draw 10 rows, with 3 columns like so.

```
> require(MASS)
> RNGkind("L'Ecuyer-CMRG")
> set.seed(12345)
> .Random.seed
```

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

```
> X0 <- MASS::mvrnorm(n=10, mu = c(0,0,0), Sigma = diag(3))
> X0
```

```
             [,1]         [,2]        [,3]
 [1,] -0.41521403 -0.415186257 -1.4578504
 [2,] -0.06116836  0.007725897 -0.4524613
 [3,] -0.15157682  1.108595682  0.3650586
 [4,] -0.27032601 -1.061102225 -1.5709113
 [5,]  0.40011478  0.501315640  1.1419086
 [6,]  1.24058838 -1.257942364  0.4615982
 [7,]  2.05311263 -1.337649667 -1.1391027
 [8,]  0.21310160  1.664438600 -0.7327706
 [9,]  0.95369382 -0.843808588  0.1703884
[10,] -0.33083472  0.470934554 -0.6220064
```

I had expected, wrongly as it turned out, that if we reduced the size of the requested sample, we would receive the first 5 rows of X0.

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

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

```
> X1 <- MASS::mvrnorm(n=5, mu = c(0,0,0), Sigma = diag(3))
> X1
```

```
             [,1]       [,2]       [,3]
[1,] -0.415186257  0.4615982 -1.4578504
[2,]  0.007725897 -1.1391027 -0.4524613
[3,]  1.108595682 -0.7327706  0.3650586
[4,] -1.061102225  0.1703884 -1.5709113
[5,]  0.501315640 -0.6220064  1.1419086
```

And I had hoped, in vain, that if I drew a larger sample, that the first 10 observations would match matrix X0.

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

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

```
> X2 <- MASS::mvrnorm(n=15, mu = c(0,0,0), Sigma = diag(3))
> X2
```

```
             [,1]        [,2]         [,3]
 [1,] -0.14581330 -1.25794236 -1.457850350
 [2,] -1.76307452 -1.33764967 -0.452461265
 [3,] -0.64267797  1.66443860  0.365058637
 [4,] -1.14302607 -0.84380859 -1.570911286
 [5,] -1.04504337  0.47093455  1.141908584
 [6,]  0.35581904 -0.41521403  0.461598191
 [7,] -0.01775994 -0.06116836 -1.139102694
 [8,] -0.51575093 -0.15157682 -0.732770619
 [9,] -0.87075954 -0.27032601  0.170388373
[10,] -0.66280180  0.40011478 -0.622006373
[11,] -1.28376925  1.24058838 -0.415186257
[12,]  0.80745894  2.05311263  0.007725897
[13,] -1.61953118  0.21310160  1.108595682
[14,] -1.31038475  0.95369382 -1.061102225
[15,] -0.39879176 -0.33083472  0.501315640
```

This is an unsatisfactory situation, of course. The first observations in the third column are the same in all three sets, while the rest differ. In a simulation exercise, this would have odd effects on our understanding of the effects of re-sampling and changes in sample size.

Only a small change in the mvrnorm code is required to solve this problem. Packaged with parallelPortableSeeds one finds a new version of mvrnorm that has better results, at least in terms of replication:

```
> library(portableParallelSeeds)
> set.seed(12345)
> Y0 <- portableParallelSeeds::mvrnorm(n=10, mu = c(0,0,0), Sigma = diag(3))
> Y0
```

```
             [,1]       [,2]        [,3]
 [1,]  0.365058637 -0.4524613 -1.45785035
 [2,]  0.461598191  1.1419086 -1.57091129
 [3,]  0.170388373 -0.7327706 -1.13910269
 [4,]  0.007725897 -0.4151863 -0.62200637
 [5,]  0.501315640 -1.0611022  1.10859568
 [6,]  1.664438600 -1.3376497 -1.25794236
 [7,] -0.415214031  0.4709346 -0.84380859
 [8,] -0.270326014 -0.1515768 -0.06116836
 [9,]  2.053112626  1.2405884  0.40011478
[10,] -0.330834724  0.9536938  0.21310160
```

```
> set.seed(12345)
> Y1 <- portableParallelSeeds::mvrnorm(n=5, mu = c(0,0,0), Sigma = diag(3))
> Y1
```

```
            [,1]       [,2]       [,3]
[1,] 0.365058637 -0.4524613 -1.4578504
[2,] 0.461598191  1.1419086 -1.5709113
[3,] 0.170388373 -0.7327706 -1.1391027
[4,] 0.007725897 -0.4151863 -0.6220064
[5,] 0.501315640 -1.0611022  1.1085957
```

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

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

```
> rnorm(1)
```

```
[1] -1.45785
```

```
> Y2 <- portableParallelSeeds::mvrnorm(n=15, mu = c(0,0,0), Sigma = diag(3))
> Y2
```

```
             [,1]         [,2]       [,3]
 [1,] -1.57091129  0.365058637 -0.4524613
 [2,] -1.13910269  0.461598191  1.1419086
 [3,] -0.62200637  0.170388373 -0.7327706
 [4,]  1.10859568  0.007725897 -0.4151863
 [5,] -1.25794236  0.501315640 -1.0611022
 [6,] -0.84380859  1.664438600 -1.3376497
 [7,] -0.06116836 -0.415214031  0.4709346
 [8,]  0.40011478 -0.270326014 -0.1515768
 [9,]  0.21310160  2.053112626  1.2405884
[10,] -0.14581330 -0.330834724  0.9536938
[11,] -1.14302607 -0.642677967 -1.7630745
[12,] -0.01775994  0.355819039 -1.0450434
[13,] -0.66280180 -0.870759537 -0.5157509
[14,] -1.61953118  0.807458942 -1.2837692
[15,] -0.05531771 -0.398791759 -1.3103848
```

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

L'Ecuyer, Pierre. 1999. "Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators." *Operations Research* 47(1):159–164. 2.2

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. 2.2

Mascagni, Michael, David Ceperley and Ashok Srinivasan. 2000. "SPRNG: A Scalable Library for Pseudorandom Number Generation." *ACM Transactions on Mathematical Software* 26:436—461. 2.1

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. 2.1

Matsumoto, Makoto and Takuji Nishimura. 1998. "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator." *ACM Trans. Model. Comput. Simul.* 8(1):3–30. 2

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