rcrandom: A pseudo-random number generator for random variables and vectors

Ricardo T. Lemos

2016-04-20

Index

Demo 1

Here you can find the basic features of this package:

- how to set up a random number generator for scalars and vectors
- how to draw continuous and discrete uniform variates
- how to reset a generator

```
library(rcrandom)
# ------------------------------------------------------------------------------
# Simplest case: one random number generator (RNG) with one stream
# ------------------------------------------------------------------------------
g <- rcrng()
# Generating 4 uniform variates with this RNG
g$runif(n = 4)
```

```
## [1] 0.08196407 0.67388213 0.41543928 0.49042872
```

```
# Example of reproducibility: resetting stream and regenerating 4 variates
g$reset()
g$runif(n = 4)
```

```
## [1] 0.08196407 0.67388213 0.41543928 0.49042872
```

```
#Generating integer uniform variates
g$rint(n = 12, lb = -3, ub = 3)
```

```
##  [1]  1 -1 -1  3  1  0 -3  2 -3 -2  3 -3
```

```
# A new generator has a different initial seed (2^127 units away from 1st one);
# thus, random numbers differ from the previous generator's.
k <- rcrng()
k$runif(n = 4)
```

```
## [1] 0.06037399 0.20826659 0.10314791 0.09759640
```

```
# A 3rd generator, for a random vector of size 2, generates 2 variates per call
h <- rcrng(2)
h$runif()
```

```
##           [,1]
## [1,] 0.9708223
## [2,] 0.1836618
```

```
# Resetting this RNG and requesting 3 pairs of variates
h$reset()
h$runif(n = 3)
```

```
##           [,1]      [,2]     [,3]
## [1,] 0.9708223 0.8970060 0.544171
## [2,] 0.1836618 0.3061345 0.331100
```

```
# ------------------------------------------------------------------------------
# Special feature of R Reference Classes: using 2 pointers to the same generator
# ------------------------------------------------------------------------------
pointer1 <- rcrng()
pointer2 <- pointer1
pointer1 #this shows the state of the RNG
```

```
## cg[ 1 ] = { 3224044943 1227141655 2220611050 1504589054 2829780440 108189859 }
```

```
pointer2
```

```
## cg[ 1 ] = { 3224044943 1227141655 2220611050 1504589054 2829780440 108189859 }
```

```
pointer1$runif(3) #generating variates with pointer1
```

```
## [1] 0.3958548 0.3994434 0.8612673
```

```
pointer1
```

```
## cg[ 1 ] = { 1279384507 2963518897 2565773750 3874168452 1247922751 3161626336 }
```

```
pointer2 #having used pointer1 to get variates also changed state of pointer2
```

```
## cg[ 1 ] = { 1279384507 2963518897 2565773750 3874168452 1247922751 3161626336 }
```

```
pointer2$runif(1) #using pointer2 also changes pointer1
```

```
## [1] 0.5042301
```

```
pointer1
```

```
## cg[ 1 ] = { 2963518897 2565773750 3672525035 1247922751 3161626336 1506873293 }
```

```
pointer2
```

```
## cg[ 1 ] = { 2963518897 2565773750 3672525035 1247922751 3161626336 1506873293 }
```

Demo 2

Here you can find some advanced features of this package:

- how to save memory by sharing the same algorithm across RNGs and/or disabling the generator's reset feature

- how to use antithetic and high precision variates
- two ways to create independent RNGs that start from the same seed
- creating lagged RNGs

```
# ------------------------------------------------------------------------------
# Memory saving strategies
# ------------------------------------------------------------------------------
# Sharing the same algorithm across RNGs and disabling the "reset button"
rcrng.globalalgorithm  <- rcmrg32k3a(name = "my.algo")
g1 <- rcrng(resettable = FALSE) #this RNG can't be reset
g2 <- rcrng()
g1$algorithm$name('new.name') #not a common procedure
rcrng.globalalgorithm$name() #name was changed
```

```
## [1] "my.algo"
```

```
g1$runif()
```

```
## [1] 0.9737495
```

```
g2$runif()
```

```
## [1] 0.9102831
```

```
# g1$reset() #would throw an error
g2$reset()
g1$runif() #g1 was not reset
```

```
## [1] 0.2507435
```

```
g2$runif() #g2 was reset
```

```
## [1] 0.9102831
```

```
# ------------------------------------------------------------------------------
# Linked pseudo-random number generators
# ------------------------------------------------------------------------------
# a) Creating 3 RNGs with the same seed but different modes
h1 <- rcrng()
h2 <- rcrng()
h2$seed(h1$seed()) # one way of setting the seed of one RNG equal to another
h3 <- h1$copy()    # this is a more convenient way to do the same thing
h2$antithetic(TRUE)
h3$high.precision(TRUE)
h1
```

```
## cg[ 1 ] = { 3984477137 1267973573 3770063761 216527865 1568537936 1200352663 }
```

```
h2
```

```
## cg[ 1 ] = { 3984477137 1267973573 3770063761 216527865 1568537936 1200352663 }
```

```
h3
```

```
## cg[ 1 ] = { 3984477137 1267973573 3770063761 216527865 1568537936 1200352663 }
```

```
h1$runif() # u
```

```
## [1] 0.556048
```

```
h2$runif() # 1 - u
```

```
## [1] 0.443952
```

```
h3$runif() # a different variate
```

```
## [1] 0.556048
```

```
h1
```

```
## cg[ 1 ] = { 1267973573 3770063761 300899854 1568537936 1200352663 2207659234 }
```

```
h2 #this stream is still aligned with h1's
```

```
## cg[ 1 ] = { 1267973573 3770063761 300899854 1568537936 1200352663 2207659234 }
```

```
h3 #this stream moves twice as fast as the previous ones
```

```
## cg[ 1 ] = { 3770063761 300899854 3472409597 1200352663 2207659234 3580718068 }
```

```r
# b) Creating lagged RNGs
k1 <- rcrng(1)
k2 <- k1$copy()
k3 <- k1$copy()
k2$advance.state(ee = 0, cc = 1) #advancing the state by 2^ee+cc = 1
k3$advance.state(ee = 1, cc = 0) #advancing by 2
k1
```

```
## cg[ 1 ] = { 3645411182 1391027122 3995023402 536114258 1376034799 2391282907 }
```

```r
k2
```

```
## cg[ 1 ] = { 1391027122 3995023402 1774831722 1376034799 2391282907 3532713983 }
```

```r
k3
```

```
## cg[ 1 ] = { 3995023402 1774831722 4062779649 2391282907 3532713983 145309005 }
```

```r
k1$runif(3)
```

```
## [1] 0.5907111 0.9121073 0.8970339
```

```r
k2$runif(3)
```

```
## [1] 0.9121073 0.8970339 0.7882744
```

```r
k3$runif(3)
```

```
## [1] 0.8970339 0.7882744 0.8868424
```

```r
# Confirming that the substream responsible for the 2nd element in the random
# vector is 2^76 (approximately 7.6*10^22) units apart from the 1st element
j <- rcrng(2)
j$runif(n = 3)
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.8068175 0.3137908 0.3238455
## [2,] 0.1082935 0.5404926 0.3600516
```

```r
j$next.substream(1)
j$reset()
j$runif(n = 3)
```

```
##           [,1]      [,2]      [,3]
## [1,] 0.1082935 0.5404926 0.3600516
## [2,] 0.1082935 0.5404926 0.3600516
```

Demo 3

Here you can find how to generate random numbers in parallel:

- Step 1. Create a local RNG
- Step 2. Initiate the cluster and export the RNG to the slave processes
- Step 3. Use sfLapply

```
library("snowfall")
sfInit(parallel = TRUE, cpus = 2)
```

```
## R Version:  R version 3.2.2 (2015-08-14)
```

```
sfLibrary(rcrandom) #slave processes load package
```

```
## Library rcrandom loaded.
```

```
# ------------------------------------------------------------------------------
# Reproducible variates generated in parallel
# ------------------------------------------------------------------------------
# Creating a local RNG
g <- rcrng(3)
sfExport("g")
# Generating 5 triplets of variates locally
g$runif(5)
```

```
##            [,1]       [,2]      [,3]      [,4]      [,5]
## [1,] 0.39565483 0.07323484 0.7207755 0.9883056 0.9235306
## [2,] 0.13112831 0.97961748 0.3399186 0.3479631 0.4158829
## [3,] 0.06683374 0.99749405 0.1974135 0.3509145 0.3192233
```

```
#generating the same 5 triplets of variates remotely (on the 2 slave processes)
sfLapply(1:2, function(i) g$runif(5))
```

```
## [[1]]
##            [,1]       [,2]      [,3]      [,4]      [,5]
## [1,] 0.39565483 0.07323484 0.7207755 0.9883056 0.9235306
## [2,] 0.13112831 0.97961748 0.3399186 0.3479631 0.4158829
## [3,] 0.06683374 0.99749405 0.1974135 0.3509145 0.3192233
##
## [[2]]
##            [,1]       [,2]      [,3]      [,4]      [,5]
## [1,] 0.39565483 0.07323484 0.7207755 0.9883056 0.9235306
## [2,] 0.13112831 0.97961748 0.3399186 0.3479631 0.4158829
## [3,] 0.06683374 0.99749405 0.1974135 0.3509145 0.3192233
```

```
# ------------------------------------------------------------------------------
# Lagged variates (lags = 0, 1, 2) generated in parallel
# ------------------------------------------------------------------------------
k <- rcrng(3)
```

```
sfExport("k")
ok <- sfLapply(0:1, function(l) lapply(1:3, function(i){
  k$advance.state(ee = 0, cc = l * (i - 1), i = i)
}))
sfLapply(1:2, function(i) k$runif(5))
```

```
## [[1]]
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.3481963 0.8708158 0.9371674 0.7006486 0.1384546
## [2,] 0.4442674 0.9020858 0.6974059 0.8524414 0.1943943
## [3,] 0.9872922 0.4463782 0.4922014 0.4329057 0.1175500
##
## [[2]]
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] 0.3481963 0.8708158 0.9371674 0.7006486 0.1384546
## [2,] 0.9020858 0.6974059 0.8524414 0.1943943 0.5798962
## [3,] 0.4922014 0.4329057 0.1175500 0.1517833 0.2490918
```

```
# ------------------------------------------------------------------------------
# Uncorrelated variates from 2 slave processes
# ------------------------------------------------------------------------------
h <- rcrng()
sfExport("h")
ok <- sfLapply(1:2, function(i) {if (i == 2) h$next.substream(); return()} )
sfLapply(1:2, function(i) h$runif(5))
```

```
## [[1]]
## [1] 0.4769520 0.1776568 0.2073639 0.2316615 0.6124618
##
## [[2]]
## [1] 0.9203269 0.6775700 0.9127677 0.7316289 0.2758339
```

```
# ------------------------------------------------------------------------------
# Terminating slave processes
# ------------------------------------------------------------------------------
sfStop()
```