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

Ricardo T. Lemos

2015-10-02

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] 0.1270111 0.3185276 0.3091860 0.8258469
```

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

```
## [1] 0.1270111 0.3185276 0.3091860 0.8258469
```

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

```
##  [1] -2  0  0 -1 -3  2  1 -1 -1 -2  1  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] 0.7595819 0.9783106 0.6851358 0.2792696
```

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

```
##           [,1]
## [1,] 0.7285098
## [2,] 0.3896315
```

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

```
##           [,1]      [,2]      [,3]
## [1,] 0.7285098 0.9655873 0.9961841
## [2,] 0.3896315 0.2968318 0.1367892
```

```r
# ------------------------------------------------------------------------------
# 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 ] = { 2338701263 1119171942 2570676563 317077452 3194180850 618832124 }
```

```r
pointer2
```

```
## cg[ 1 ] = { 2338701263 1119171942 2570676563 317077452 3194180850 618832124 }
```

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

```
## [1] 0.09570262 0.66287062 0.23642839
```

```r
pointer1
```

```
## cg[ 1 ] = { 3074696362 2026783554 4111223072 2663656755 3474743153 3095770918 }
```

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

```
## cg[ 1 ] = { 3074696362 2026783554 4111223072 2663656755 3474743153 3095770918 }
```

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

```
## [1] 0.8299882
```

```r
pointer1
```

```
## cg[ 1 ] = { 2026783554 4111223072 3516035438 3474743153 3095770918 4246230638 }
```

```
pointer2
```

```
## cg[ 1 ] = { 2026783554 4111223072 3516035438 3474743153 3095770918 4246230638 }
```

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] 0.9053621
```

```
g2$runif()
```

```
## [1] 0.3304994
```

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

```
## [1] 0.3261578
```

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

```
## [1] 0.3304994
```

```
# ------------------------------------------------------------------------------
# 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 ] = { 796079799 2105258207 955365076 2923159030 4116632677 3067683584 }
```

```
h2
```

```
## cg[ 1 ] = { 796079799 2105258207 955365076 2923159030 4116632677 3067683584 }
```

```
h3
```

```
## cg[ 1 ] = { 796079799 2105258207 955365076 2923159030 4116632677 3067683584 }
```

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

```
## [1] 0.968134
```

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

```
## [1] 0.03186595
```

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

```
## [1] 0.9681341
```

```
h1
```

```
## cg[ 1 ] = { 2105258207 955365076 133908661 4116632677 3067683584 270771878 }
```

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

```
## cg[ 1 ] = { 2105258207 955365076 133908661 4116632677 3067683584 270771878 }
```

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

```
## cg[ 1 ] = { 955365076 133908661 2014066392 3067683584 270771878 971442415 }
```

```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 ] = { 4215590817 3862461878 1087200967 1544910132 936383720 1611370123 }
```

```r
k2
```

```
## cg[ 1 ] = { 3862461878 1087200967 354555486 936383720 1611370123 3392835665 }
```

```r
k3
```

```
## cg[ 1 ] = { 1087200967 354555486 467152754 1611370123 3392835665 3218863532 }
```

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

```
## [1] 0.2925952 0.3593174 0.2368010
```

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

```
## [1] 0.35931738 0.23680101 0.05839683
```

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

```
## [1] 0.23680101 0.05839683 0.13123030
```

```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,] 0.9575618 0.09600628 0.09794506
## [2,] 0.2355107 0.84743100 0.65011277
```

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

```
##           [,1]     [,2]      [,3]
## [1,] 0.2355107 0.847431 0.6501128
## [2,] 0.2355107 0.847431 0.6501128
```

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,] 0.4490632 0.04872808 0.26874415 0.3046164 0.5294247
## [2,] 0.1124740 0.97369279 0.05954777 0.4943962 0.5542705
## [3,] 0.8017637 0.60887316 0.06725361 0.9400301 0.3318539
```

```
#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,] 0.4490632 0.04872808 0.26874415 0.3046164 0.5294247
## [2,] 0.1124740 0.97369279 0.05954777 0.4943962 0.5542705
## [3,] 0.8017637 0.60887316 0.06725361 0.9400301 0.3318539
##
## [[2]]
##           [,1]       [,2]       [,3]      [,4]      [,5]
## [1,] 0.4490632 0.04872808 0.26874415 0.3046164 0.5294247
## [2,] 0.1124740 0.97369279 0.05954777 0.4943962 0.5542705
## [3,] 0.8017637 0.60887316 0.06725361 0.9400301 0.3318539
```

```
# ------------------------------------------------------------------------------
# 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,] 0.3183044 0.5776002 0.93884942 0.8666665 0.02536742
## [2,] 0.4444124 0.3580575 0.07770452 0.1949823 0.14771897
## [3,] 0.7432606 0.7708692 0.72433360 0.5277614 0.64656963
##
## [[2]]
##           [,1]       [,2]      [,3]      [,4]       [,5]
## [1,] 0.3183044 0.57760022 0.9388494 0.8666665 0.02536742
## [2,] 0.3580575 0.07770452 0.1949823 0.1477190 0.09898422
## [3,] 0.7243336 0.52776136 0.6465696 0.8581686 0.30106962
```

```
# ------------------------------------------------------------------------------
# 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] 0.08196407 0.67388213 0.41543928 0.49042872 0.67294595
##
## [[2]]
## [1] 0.32686001 0.05087654 0.34017910 0.01212670 0.74058980
```

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