

Using the `doRNG` package

doRNG package – Version 1.8.3

Renaud Gaujoux

November 7, 2024

Contents

Introduction	1	5 Nested and conditional loops .	10
1 The <code>%dorng%</code> operator	3	5.1 Nested loops	10
1.1 How it works	3	5.2 Conditional loops	11
1.2 Seeding computations	4	5.3 Nested conditional loops	12
1.3 Difference between <code>set.seed</code> and <code>.options.RNG</code>	6	6 Performance overhead	13
2 Parallel environment independence	7	7 Known issues	14
3 Reproducible <code>%dopar%</code> loops	8	8 News and changes	14
4 Reproducible sets of loops	9	Session information	17
		References	17

Introduction

Research reproducibility is an issue of concern, e.g. in bioinformatics [3, 9, 4]. Some analyses require multiple independent runs to be performed, or are amenable to a split-and-reduce scheme. For example, some optimisation algorithms are run multiple times from different random starting points, and the result that achieves the least approximation error is selected. The *foreach* package¹ [7] provides a very convenient way to perform parallel computations, with different parallel environments such as MPI or Redis, using a transparent loop-like syntax:

¹<https://cran.r-project.org/package=foreach>

```

# load and register parallel backend for multicore computations
library(doParallel)

## Loading required package: foreach
## Loading required package: iterators
## Loading required package: parallel

cl <- makeCluster(2)
registerDoParallel(cl)

# perform 5 tasks in parallel
x <- foreach(i=1:5) %dopar% {
  i + runif(1)
}
unlist(x)

## [1] 1.023274 2.121151 3.920280 4.397149 5.429190

```

For each parallel environment a *backend* is implemented as a specialised `%dopar%` operator, which performs the setup and pre/post-processing specifically required by the environment (e.g. export of variable to each worker). The `foreach` function and the `%dopar%` operator handle the generic parameter dispatch when the task are split between worker processes, as well as the reduce step – when the results are returned to the master worker.

When stochastic computations are involved, special random number generators must be used to ensure that the separate computations are indeed statistically independent – unless otherwise wanted – and that the loop is reproducible. In particular, standard `%dopar%` loops are not reproducible:

```

# with standard %dopar%: foreach loops are not reproducible
set.seed(123)
res <- foreach(i=1:5) %dopar% { runif(3) }
set.seed(123)
res2 <- foreach(i=1:5) %dopar% { runif(3) }
identical(res, res2)

## [1] FALSE

```

A random number generator commonly used to achieve reproducibility is the combined multiple-recursive generator from L’Ecuyer [5]. This generator can generate independent random streams, from a 6-length numeric seed. The idea is then to generate a sequence of random stream of the same length as the number of iteration (i.e. tasks) and use a different stream when computing each one of them.

The *doRNG* package² [2] provides convenient ways to implement reproducible parallel `foreach` loops, independently of the parallel backend used to perform the computation. We illustrate its use, showing how non-reproducible loops can be made reproducible, even when tasks are not scheduled in the same way in two separate set of runs, e.g. when the

²<https://cran.r-project.org/package=doRNG>

workers do not get to compute the same number of tasks or the number of workers is different. The package has been tested with the *doParallel*³ and *doMPI*⁴ packages [10, 1], but should work with other backends such as provided by the *doRedis* package⁵ [6].

1 The %dorng% operator

The *doRNG* package defines a new generic operator, %dorng%, to be used with foreach loops, instead of the standard %dopar%. Loops that use this operator are *de facto* reproducible.

```
# load the doRNG package
library(doRNG)

## Loading required package: rngtools

# using %dorng%: loops _are_ reproducible
set.seed(123)
res <- foreach(i=1:5) %dorng% { runif(3) }
set.seed(123)
res2 <- foreach(i=1:5) %dorng% { runif(3) }
identical(res, res2)

## [1] TRUE
```

1.1 How it works

For a loop with N iterations, the %dorng% operator internally performs the following tasks:

1. generate a sequence of random seeds $(S_i)_{1 \leq i \leq N}$ for the R random number generator "L'Ecuyer-CMRG" [5], using the function `nextRNGStream` from the *parallel* package⁶ [8], which ensure the different RNG streams are statistically independent;
2. modify the loop's R expression so that the random number generator is set to "L'Ecuyer-CMRG" at the beginning of each iteration, and is seeded with consecutive seeds in (S_n) : iteration i is seeded with S_i , $1 \leq i \leq N$;
3. call the standard %dopar% operator, which in turn calls the relevant (i.e. registered) foreach parallel backend;
4. store the whole sequence of random seeds as an attribute in the result object:

³<https://cran.r-project.org/package=doParallel>

⁴<https://cran.r-project.org/package=doMPI>

⁵<https://cran.r-project.org/package=doRedis>

⁶<https://cran.r-project.org/package=parallel>

```

attr(res, 'rng')

## [[1]]
## [1]      10407      642048078      81368183 -2093158836      506506973      1421492218 -1906381517
##
## [[2]]
## [1]      10407     1340772676 -1389246211 -999053355 -953732024  1888105061  2010658538
##
## [[3]]
## [1]      10407 -1318496690 -948316584   683309249 -990823268 -1895972179  1275914972
##
## [[4]]
## [1]      10407      524763474  1715794407  1887051490 -1833874283   494155061 -1221391662
##
## [[5]]
## [1]      10407 -1816009034 -580124020  1603250023   817712173   190009158 -706984535

```

1.2 Seeding computations

Sequences of random streams for "L'Ecuyer-CMRG" are generated using a 6-length integer seed, e.g.,:

```

nextRNGStream(c(407L, 1:6))

## [1]      407 -447371532  542750874 -935969228 -269326340  701604884 -1748056907

```

However, the `%dorng%` operator provides alternative – convenient – ways of seeding reproducible loops.

set.seed: as shown above, calling `set.seed` before the loop ensure reproducibility of the results, using a single integer as a seed. The actual 6-length seed is then generated with an internal call to `RNGkind("L'Ecuyer-CMRG")`.

.options.RNG with single integer: the `%dorng%` operator support options that can be passed in the `foreach` statement, containing arguments for the internal call to `set.seed`:

```

# use a single numeric as a seed
s <- foreach(i=1:5, .options.RNG=123) %dorng% { runif(3) }
s2 <- foreach(i=1:5, .options.RNG=123) %dorng% { runif(3) }
identical(s, s2)

## [1] TRUE

```

Note: calling `set.seed` before the loop is equivalent to passing the seed in `.options.RNG`. See [Section 1.3](#) for more details.

The kind of Normal generator may also be passed in `.options.RNG`:

```

## Pass the Normal RNG kind to use within the loop
# results are identical if not using the Normal kind in the loop
optsN <- list(123, normal.kind="Ahrens")
resN.U <- foreach(i=1:5, .options.RNG=optsN) %dorng% { runif(3) }
identical(resN.U[1:5], res[1:5])

## [1] TRUE

# Results are different if the Normal kind is used and is not the same
resN <- foreach(i=1:5, .options.RNG=123) %dorng% { rnorm(3) }
resN1 <- foreach(i=1:5, .options.RNG=optsN) %dorng% { rnorm(3) }
resN2 <- foreach(i=1:5, .options.RNG=optsN) %dorng% { rnorm(3) }
identical(resN[1:5], resN1[1:5])

## [1] FALSE

identical(resN1[1:5], resN2[1:5])

## [1] TRUE

```

.options.RNG with 6-length: the actual 6-length integer seed used for the first RNG stream may be passed via `options.RNG`:

```

# use a 6-length numeric
s <- foreach(i=1:5, .options.RNG=1:6) %dorng% { runif(3) }
attr(s, 'rng')[1:3]

## [[1]]
## [1] 10407      1      2      3      4      5      6
##
## [[2]]
## [1]      10407 -447371532  542750874 -935969228 -269326340  701604884 -1748056907
##
## [[3]]
## [1]      10407  311773008 -1393648596  433058656 -545474683  2059732357  994549473

```

.options.RNG with 7-length: a 7-length integer seed may also be passed via `options.RNG`, which is useful to seed a loop with the value of `.Random.seed` as used in some iteration of another loop⁷:

```

# use a 7-length numeric, used as first value for .Random.seed
seed <- attr(res, 'rng')[[2]]
s <- foreach(i=1:5, .options.RNG=seed) %dorng% { runif(3) }
identical(s[1:4], res[2:5])

```

⁷Note that the RNG kind is then always required to be the "L'Ecuyer-CMRG", i.e. the first element of the seed must have unit 7 (e.g. 407 or 107).

```
## [1] TRUE
```

.options.RNG with complete sequence of seeds: the complete description of the sequence of seeds to be used may be passed via `options.RNG`, as a list or a matrix with the seeds in columns. This is useful to seed a loop exactly as desired, e.g. using an RNG other than "L'Ecuyer-CMRG", or using different RNG kinds in each iteration, which probably have different seed length, in order to compare their stochastic properties. It also allows to reproduce `%dorng%` loops without knowing their seeding details:

```
# reproduce previous %dorng% loop
s <- foreach(i=1:5, .options.RNG=res) %dorng% { runif(3) }
identical(s, res)

## [1] TRUE

## use completely custom sequence of seeds (e.g. using RNG "Marsaglia-Multicarry")
# as a matrix
seedM <- rbind(rep(401, 5), mapply(rep, 1:5, 2))
seedM

##      [,1] [,2] [,3] [,4] [,5]
## [1,] 401  401  401  401  401
## [2,]   1    2    3    4    5
## [3,]   1    2    3    4    5

sM <- foreach(i=1:5, .options.RNG=seedM) %dorng% { runif(3) }
# same seeds passed as a list
seedL <- lapply(seq(ncol(seedM)), function(i) seedM[,i])
sL <- foreach(i=1:5, .options.RNG=seedL) %dorng% { runif(3) }
identical(sL, sM)

## [1] TRUE
```

1.3 Difference between `set.seed` and `.options.RNG`

While it is equivalent to seed `%dorng%` loops with `set.seed` and `.options.RNG`, it is important to note that the result depends on the current RNG kind ⁸:

```
# default RNG kind
RNGkind('default')
def <- foreach(i=1:5, .options.RNG=123) %dorng% { runif(3) }
```

⁸See [Section 7](#) about a bug in versions ≥ 1.4 on this feature.

```

# Marsaglia-Multicarry
RNGkind('Marsaglia')

## Warning in RNGkind("Marsaglia"): RNGkind: Marsaglia-Multicarry has poor statistical
properties

mars <- foreach(i=1:5, .options.RNG=123) %dorng% { runif(3) }
identical(def, mars)

## [1] FALSE

# revert to default RNG kind
RNGkind('default')

```

This is a “normal” behaviour, which is a side-effect of the expected equivalence between `set.seed` and `.options.RNG`. This should not be a problem for reproducibility though, as R RNGs are stable across versions, and loops are most of the time used with the default RNG settings. In order to ensure seeding is independent from the current RNG, one has to pass a 7-length numeric seed to `.options.RNG`, which is then used directly as a value for `.Random.seed` (see below).

2 Parallel environment independence

An important feature of `%dorng%` loops is that their result is independent of the underlying parallel physical settings. Two separate runs seeded with the same value will always produce the same results. Whether they use the same number of worker processes, parallel backend or task scheduling does not influence the final result. This also applies to computations performed sequentially with the `doSEQ` backend. The following code illustrates this feature using 2 or 3 workers.

```

# define a stochastic task to perform
task <- function() c(pid=Sys.getpid(), val=runif(1))

# using the previously registered cluster with 2 workers
set.seed(123)
res_2workers <- foreach(i=1:5, .combine=rbind) %dorng% {
  task()
}
# stop cluster
stopCluster(cl)

# Sequential computation
registerDoSEQ()
set.seed(123)
res_seq <- foreach(i=1:5, .combine=rbind) %dorng% {
  task()
}

```

```

}
#

# Using 3 workers
# NB: if re-running this vignette you should edit to force using 3 here
cl <- makeCluster( if(isManualVignette()) 3 else 2)
length(cl)

## [1] 2

# register new cluster
registerDoParallel(cl)
set.seed(123)
res_3workers <- foreach(i=1:5, .combine=rbind) %dorng% {
  task()
}
# task schedule is different
pid <- rbind(res1=res_seq[,1], res_2workers[,1], res2=res_3workers[,1])
storage.mode(pid) <- 'integer'
pid

##      result.1 result.2 result.3 result.4 result.5
## res1      5840      5840      5840      5840      5840
##          5887      5888      5887      5888      5887
## res2      6015      6014      6015      6014      6015

# results are identical
identical(res_seq[,2], res_2workers[,2]) && identical(res_2workers[,2], res_3workers[,2])

## [1] TRUE

```

3 Reproducible %dopar% loops

The *doRNG* package also provides a non-invasive way to convert %dopar% loops into reproducible loops, i.e. without changing their actual definition. It is useful to quickly ensure the reproducibility of existing code or functions whose definition is not accessible (e.g. from other packages). This is achieved by registering the doRNG backend:

```

set.seed(123)
res <- foreach(i=1:5) %dorng% { runif(3) }

registerDoRNG(123)
res_dopar <- foreach(i=1:5) %dopar% { runif(3) }
identical(res_dopar, res)

## [1] TRUE

attr(res_dopar, 'rng')

```

```
## [[1]]
## [1]      10407      642048078      81368183 -2093158836      506506973      1421492218 -1906381517
##
## [[2]]
## [1]      10407     1340772676 -1389246211 -999053355 -953732024  1888105061  2010658538
##
## [[3]]
## [1]      10407 -1318496690 -948316584   683309249 -990823268 -1895972179  1275914972
##
## [[4]]
## [1]      10407     524763474  1715794407  1887051490 -1833874283  494155061 -1221391662
##
## [[5]]
## [1]      10407 -1816009034 -580124020  1603250023   817712173  190009158 -706984535
```

4 Reproducible sets of loops

Sequences of multiple loops are reproducible, whether using the `%dorng%` operator or the registered `doRNG` backend:

```
set.seed(456)
s1 <- foreach(i=1:5) %dorng% { runif(3) }
s2 <- foreach(i=1:5) %dorng% { runif(3) }
# the two loops do not use the same streams: different results
identical(s1, s2)

## [1] FALSE

# but the sequence of loops is reproducible as a whole
set.seed(456)
r1 <- foreach(i=1:5) %dorng% { runif(3) }
r2 <- foreach(i=1:5) %dorng% { runif(3) }
identical(r1, s1) && identical(r2, s2)

## [1] TRUE

# one can equivalently register the doRNG backend and use %dopar%
registerDoRNG(456)
r1 <- foreach(i=1:5) %dopar% { runif(3) }
r2 <- foreach(i=1:5) %dopar% { runif(3) }
identical(r1, s1) && identical(r2, s2)

## [1] TRUE
```

5 Nested and conditional loops

Nested and conditional foreach loops are currently not supported and generate an error:

```
# nested loop
try( foreach(i=1:10) %:% foreach(j=1:i) %dorng% { rnorm(1) } )

## Error: nested/conditional foreach loops are not supported yet.
## See the package's vignette for a work around.

# conditional loop
try( foreach(i=1:10) %:% when(i %% 2 == 0) %dorng% { rnorm(1) } )

## Error: nested/conditional foreach loops are not supported yet.
## See the package's vignette for a work around.
```

In this section, we propose a general work around for this kind of loops, that will eventually be incorporated in the `%dorng%` operator – when I find out how to mimic its behaviour from the operator itself.

5.1 Nested loops

The idea is to create a sequence of RNG seeds before the outer loop, and use each of them successively to set the RNG in the inner loop – which is exactly what `%dorng%` does for simple loops:

```
# doRNG must not be registered
registerDoParallel(cl)

# generate sequence of seeds of length the number of computations
n <- 10; p <- 5
rng <- RNGseq( n * p, 1234)

# run standard nested foreach loop
res <- foreach(i=1:n) %:% foreach(j=1:p, r=rng[(i-1)*p + 1:p]) %dopar% {

  # set RNG seed
  rngtools::setRNG(r)

  # do your own computation ...
  c(i, j, rnorm(1))
}

# Compare against the equivalent sequential computations
k <- 1
res2 <- foreach(i=1:n) %:% foreach(j=1:p) %do%{
  # set seed
  rngtools::setRNG(rng[[k]])
  k <- k + 1
}
```

```

    # do your own computation ...
    c(i, j, rnorm(1))
}

stopifnot( identical(res, res2) )

```

The following is a more complex example with unequal – but *known a priori* – numbers of iterations performed in the inner loops:

```

# generate sequence of seeds of length the number of computations
n <- 10
rng <- RNGseq( n * (n+1) / 2, 1234)

# run standard nested foreach loop
res <- foreach(i=1:n) %:% foreach(j=1:i, r=rng[(i-1)*i/2 + 1:i]) %dopar%{

    # set RNG seed
    rngtools::setRNG(r)

    # do your own computation ...
    c(i, j, rnorm(1))
}

# Compare against the equivalent sequential computations
k <- 1
res2 <- foreach(i=1:n) %:% foreach(j=1:i) %do%{
    # set seed
    rngtools::setRNG(rng[[k]])
    k <- k + 1

    # do your own computation ...
    c(i, j, rnorm(1))
}

stopifnot( identical(res, res2) )

```

5.2 Conditional loops

The work around used for nested loops applies to conditional loops that use the `when()` clause. It ensures that the RNG seed use for a given inner iteration does not depend on the filter, but only on its index in the unconditional-unfolded loop:

```

# un-conditional single loop
resAll <- foreach(i=1:n, .options.RNG=1234) %dorng%{
    # do your own computation ...
    c(i, rnorm(1))
}

```

```

# generate sequence of RNG
rng <- RNGseq(n, 1234)

# conditional loop: even iterations
resEven <- foreach(i=1:n, r=rng) %:% when(i %% 2 == 0) %dopar%{

  # set RNG seed
  rngtools::setRNG(r)

  # do your own computation ...
  c(i, rnorm(1))
}

# conditional loop: odd iterations
resOdd <- foreach(i=1:n, r=rng) %:% when(i %% 2 == 1) %dopar%{

  # set RNG seed
  rngtools::setRNG(r)

  # do your own computation ...
  c(i, rnorm(1))
}

# conditional loop: only first 2 and last 2
resFL <- foreach(i=1:n, r=rng) %:% when(i %in% c(1,2,n-1,n)) %dopar%{

  # set RNG seed
  rngtools::setRNG(r)

  # do your own computation ...
  c(i, rnorm(1))
}

# compare results
stopifnot( identical(resAll[seq(2,n,by=2)], resEven) )
stopifnot( identical(resAll[seq(1,n,by=2)], resOdd) )
stopifnot( identical(resAll[c(1,2,n-1,n)], resFL) )

```

5.3 Nested conditional loops

Conditional nested loops may use the same work around, as shown in this intricate example:

```

# generate sequence of seeds of length the number of computations
n <- 10
rng <- RNGseq( n * (n+1) / 2, 1234)

# run standard nested foreach loop
res <- foreach(i=1:n) %:% when(i %% 2 == 0) %:% foreach(j=1:i, r=rng[(i-1)*i/2 + 1:i]) %dopar%{

```

```

    # set RNG seed
    rngtools::setRNG(r)

    # do your own computation ...
    c(i, j, rnorm(1))
}

# Compare against the equivalent sequential computations
k <- 1
resAll <- foreach(i=1:n) %:% foreach(j=1:i) %do%{
  # set seed
  rngtools::setRNG(rng[[k]])
  k <- k + 1

  # do your own computation ...
  c(i, j, rnorm(1))
}

stopifnot( identical(resAll[seq(2,n,by=2)], res) )

```

6 Performance overhead

The extra setup performed by the `%dorng%` operator leads to a slight performance overhead, which might be significant for very quick computations, but should not be a problem for realistic computations. The benchmarks below show that a `%dorng%` loop may take up to two seconds more than the equivalent `%dopar%` loop, which is not significant in practice, where parallelised computations typically take several minutes.

```

# load rbenchmark
library(rbenchmark)

# comparison is done on sequential computations
registerDoSEQ()
rPar <- function(n, s=0){ foreach(i=1:n) %dopar% { Sys.sleep(s) } }
rRNG <- function(n, s=0){ foreach(i=1:n) %dorng% { Sys.sleep(s) } }

# run benchmark
cmp <- benchmark(rPar(10), rRNG(10)
                , rPar(25), rRNG(25)
                , rPar(50), rRNG(50)
                , rPar(50, .01), rRNG(50, .01)
                , rPar(10, .05), rRNG(10, .05)
                , replications=5)

# order by increasing elapsed time
cmp[order(cmp$elapsed), ]

##          test replications elapsed relative user.self sys.self user.child sys.child

```

## 1	rPar(10)	5	0.011	1.000	0.011	0.000	0	0
## 2	rRNG(10)	5	0.020	1.818	0.020	0.000	0	0
## 3	rPar(25)	5	0.020	1.818	0.020	0.000	0	0
## 4	rRNG(25)	5	0.033	3.000	0.033	0.000	0	0
## 5	rPar(50)	5	0.034	3.091	0.034	0.000	0	0
## 6	rRNG(50)	5	0.056	5.091	0.056	0.000	0	0
## 9	rPar(10, 0.05)	5	2.532	230.182	0.029	0.000	0	0
## 10	rRNG(10, 0.05)	5	2.544	231.273	0.039	0.000	0	0
## 7	rPar(50, 0.01)	5	2.594	235.818	0.071	0.004	0	0
## 8	rRNG(50, 0.01)	5	2.627	238.818	0.108	0.000	0	0

7 Known issues

- Nested and/or conditional foreach loops using the operator `:%` are not currently not supported (see [Section 5](#) for a work around).
- An error is thrown in `doRNG` 1.2.6, when the package `iterators` was not loaded, when used with `foreach i= 1.4`.
- There was a bug in versions prior to 1.4, which caused `set.seed` and `.options.RNG` not to be equivalent when the current RNG was "L'Ecuyer-CMRG". This behaviour can still be reproduced by setting:

```
doRNGversion('1.3')
```

To revert to the latest default behaviour:

```
doRNGversion(NULL)
```

8 News and changes

```
*****
Changes in version 1.8
*****
CHANGES
  o Unit tests are now run through testthat
  o Minor fixes in man pages and README file
  o Now depends on rngtools >= 1.3
  o The result list gains an attribute 'doRNG_version' that contains
    the version of doRNG that was used, based on doRNGversion().
  NB: this is not necessarily the same as the version of the installed
    package.
  o Added the following global option 'doRNG.rng_change_warning_skip'.
  See ?`%dornng%` (issue #14).
  o Moved dependency on pkgmaker to Suggests to make installation lighter
    (issue #10).

FIXES
  o Enabled running %dornng% loops within a package (incorporating
```

- the solution proposed by Elizabeth Byerly in PR#3)
- o Fixed error with NULL iteration results when setting 'rng' attribute (issue #9)
 - o Fixed error when using unnamed foreach arguments (issue #8)
 - o Fixed non-reproducibility issue when the .Random.seed is not yet initialized, e.g., when the session starts and the RNG has not been used yet (issue #12)
 - o Fixed runtime error when package is not attached (issue #13)

 Changes in version 1.6.2

FIXES

- o Non reproducible %dornrg% loop when doRNG is registered over doSEQ (Issue #1 reported by Brenton Kenkel). Actually due to %dornrg% not restoring the RNG (to state + 1) when doRNG is registered over doSEQ.
- o %dornrg% was not working properly on loops of length one (Issue #2)

 Changes in version 1.6

CHANGES

- o doRNG now depends on the package pkgmaker (>= 0.20)

FIXES

- o Check error due number of cores used. Now limited to 2 in examples, vignette and unit test.

 Changes in version 1.5

CHANGES

- o doRNG now depends on the package pkgmaker (>= 0.9)
- o improved vignette
- o most of the general RNG utilities have been incorporated in a new package called rngtools.

 Changes in version 1.4.1

CHANGES

- o when the current RNG was L'Ecuyer-CMRG, unseeded loops now use the current RNG stream as for the first stream in the RNG sequence and changes the current RNG to the next RNG stream of the last stream in the sequence.

BUG FIX

- o fix error "'iter' not found" due to changes in foreach package dependencies -- that was announced by Rich Calaway.
- o loops seeded with set.seed and .options.RNG were not reproducible when current RNG was L'Ecuyer-CMRG (reported by Zhang Peng)
- o separate unseeded loops were sharing most of their streams, when current RNG was L'Ecuyer-CMRG the RNG seed.
- o nested/conditional loops were crashing with a bad error. They are still not supported but the error message is nicer and a work around has been added to the vignette (reported by Chanhee Yi and Zhang Peng).

 Changes in version 1.2.3

BUG FIX

- o fixed error when running a %dorng% loop on a fresh session, with no parallel backend registered.

CHANGES

- o improved vignette
- o added more unit tests
- o changed the name of the RNG attribute on result of %dorng% loops from 'RNG' to 'rng'. It now contains the whole sequence of RNG seeds, instead of only the first one.
- o RNGseq now accepts a list or a matrix describing the whole sequence of seeds. See vignette for more details.
- o %dorng% loops can be seeded with a complete sequence of seeds passed as a list, a matrix, or an object with attribute 'rng', e.g. the results of %dorng% loops. See vignette for more details.

Changes in version 1.2.2

BUG FIX

- o separate %dorng% loops were using the same seed.

NEW FEATURES

- o add unit tests
- o first seed is set as an attribute of the loop's result

CHANGES

- o function doRNGseed now returns the seed to use for the first iteration.
- o RNGseq now change the current RNG state if called with no seed specific.

DEFUNCT

- o removed function CMRGseed

Changes in version 1.2

BUG FIX

- o An error was thrown if using %dorng% loops before using any random generator. Thanks to Eric Lehmann for reporting this.

CHANGES

- o add vignette
- o use package doParallel in examples

Changes in version 1.1

CHANGES

- o use R core RNG "L'Ecuyer-CMRG" and the parallel package, instead of the implementation provided by the rstream package.

Cleanup

```
stopCluster(c1)
```

Session information

```
R version 4.4.2 (2024-10-31)
Platform: x86_64-pc-linux-gnu
Running under: Ubuntu 24.04.1 LTS

Matrix products: default
BLAS: /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblas-p-r0.3.26.so; LAPACK version 3.12.0

Random number generation:
RNG: L'Ecuyer-CMRG
Normal: Inversion
Sample: Rejection

locale:
 [1] LC_CTYPE=en_US.UTF-8      LC_NUMERIC=C              LC_TIME=en_US.UTF-8
 [4] LC_COLLATE=C             LC_MONETARY=en_US.UTF-8  LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8     LC_NAME=C                 LC_ADDRESS=C
[10] LC_TELEPHONE=C          LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C

time zone: Etc/UTC
tzcode source: system (glibc)

attached base packages:
[1] parallel stats graphics grDevices utils datasets methods base

other attached packages:
[1] rbenchmark_1.0.0 doRNG_1.8.3          rngtools_1.5.3      doParallel_1.0.17
[5] iterators_1.0.14 foreach_1.5.2        knitr_1.48          pkgmaker_0.32.10
[9] registry_0.5-1

loaded via a namespace (and not attached):
 [1] vctrs_0.6.5      cli_3.6.3          rlang_1.1.4        xfun_0.49          stringi_1.8.4
 [6] highr_0.11       assertthat_0.2.1  xtable_1.8-4       glue_1.8.0         buildtools_1.0.0
[11] maketools_1.3.1  sys_3.4.3          evaluate_1.0.1     lifecycle_1.0.4   stringr_1.5.1
[16] compiler_4.4.2  codetools_0.2-20  digest_0.6.37     magrittr_2.0.3    tools_4.4.2
[21] withr_3.0.2
```

References

- [1] Microsoft Corporation and Steve Weston. *doParallel: Foreach Parallel Adaptor for the 'parallel' Package*. R package version 1.0.17. 2022. URL: <https://github.com/RevolutionAnalytics/doparallel>.

- [2] Renaud Gaujoux. *doRNG: Generic Reproducible Parallel Backend for 'foreach' Loops*. R package version 1.8.3. URL: <https://renozao.github.io/doRNG>.
- [3] Torsten Hothorn and Friedrich Leisch. “Case studies in reproducibility.” In: *Briefings in bioinformatics* (2011). ISSN: 1477-4054. DOI: [10.1093/bib/bbq084](https://doi.org/10.1093/bib/bbq084). URL: <http://www.ncbi.nlm.nih.gov/pubmed/21278369>.
- [4] John P A Ioannidis et al. “The reproducibility of lists of differentially expressed genes in microarray studies”. In: *Nature Genetics* 41.2 (2008), pp. 149–155. ISSN: 10614036. DOI: [10.1038/ng.295](https://doi.org/10.1038/ng.295). URL: <http://www.nature.com/doifinder/10.1038/ng.295>.
- [5] Pierre L’Ecuyer. “Good parameters and implementations for combined multiple recursive random number generators”. In: *Operations Research* 47.1 (1999). DOI: [10.1287/opre.47.1.159](https://doi.org/10.1287/opre.47.1.159). URL: <http://www.jstor.org/stable/10.2307/222902>.
- [6] B. W. Lewis. *doRedis: 'Foreach' Parallel Adapter Using the 'Redis' Database*. R package version 3.0.2. 2023.
- [7] Microsoft and Steve Weston. *foreach: Provides Foreach Looping Construct*. R package version 1.5.2. 2022. URL: <https://github.com/RevolutionAnalytics/foreach>.
- [8] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2024. URL: <https://www.R-project.org/>.
- [9] Victoria C Stodden. *The Digitization of Science: Reproducibility and Interdisciplinary Knowledge Transfer*. 2011. URL: <http://aaas.confex.com/aaas/2011/webprogram/Session3166.html>.
- [10] Steve Weston. *doMPI: Foreach Parallel Adaptor for the Rmpi Package*. R package version 0.2.2. 2017.