

# Package: dqrng (via r-universe)

September 22, 2026

**Type** Package

**Title** Fast Pseudo Random Number Generators

**Version** 0.4.1.1

**Description** Several fast random number generators are provided as C++ header only libraries: The PCG family by O'Neill (2014 <<https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf>>) as well as the Xoroshiro / Xoshiro family by Blackman and Vigna (2021 <[doi:10.1145/3460772](https://doi.org/10.1145/3460772)>). In addition fast functions for generating random numbers according to a uniform, normal and exponential distribution are included. The latter two use the Ziggurat algorithm originally proposed by Marsaglia and Tsang (2000, <[doi:10.18637/jss.v005.i08](https://doi.org/10.18637/jss.v005.i08)>). The fast sampling methods support unweighted sampling both with and without replacement. These functions are exported to R and as a C++ interface and are enabled for use with the default 64 bit generator from the PCG family, Xoroshiro128+/+/\*\* and Xoshiro256+/+/\*\* as well as the 64 bit version of the 20 rounds Threefry engine (Salmon et al., 2011, <[doi:10.1145/2063384.2063405](https://doi.org/10.1145/2063384.2063405)>) as provided by the package 'sitmo'.

**License** AGPL-3

**Depends** R (>= 3.5.0)

**Imports** Rcpp (>= 0.12.16)

**LinkingTo** Rcpp, BH (>= 1.64.0-1), sitmo (>= 2.0.0)

**RoxygenNote** 7.3.2

**Suggests** BH, testthat, knitr, rmarkdown, mvtnorm (>= 1.2-3), bench, sitmo

**VignetteBuilder** knitr

**URL** <https://daqana.github.io/dqrng/>, <https://github.com/daqana/dqrng>

**BugReports** <https://github.com/daqana/dqrng/issues>

**Encoding** UTF-8

**Config/build/compilation-database** true

**Repository** <https://rstub.r-universe.dev>  
**Date/Publication** 2025-06-29 19:55:18 UTC  
**RemoteUrl** <https://github.com/daqana/dqrng>  
**RemoteRef** HEAD  
**RemoteSha** a47400129134ed514fd469c798a4d2359b5ee84c

Contents

|                               |           |
|-------------------------------|-----------|
| dqrng-package . . . . .       | 2         |
| dqrmvnorm . . . . .           | 3         |
| dqRNGkind . . . . .           | 4         |
| dqsample . . . . .            | 6         |
| generateSeedVectors . . . . . | 7         |
| register_methods . . . . .    | 8         |
| <b>Index</b>                  | <b>10</b> |

---

|               |                                                    |
|---------------|----------------------------------------------------|
| dqrng-package | <i>dqrng: Fast Pseudo Random Number Generators</i> |
|---------------|----------------------------------------------------|

---

Description

Several fast random number generators are provided as C++ header only libraries: The PCG family by O’Neill (2014 <https://www.cs.hmc.edu/tr/hmc-cs-2014-0905.pdf>) as well as the Xoroshiro / Xoshiro family by Blackman and Vigna (2021 [doi:10.1145/3460772](https://doi.org/10.1145/3460772)). In addition fast functions for generating random numbers according to a uniform, normal and exponential distribution are included. The latter two use the Ziggurat algorithm originally proposed by Marsaglia and Tsang (2000, [doi:10.18637/jss.v005.i08](https://doi.org/10.18637/jss.v005.i08)). The fast sampling methods support unweighted sampling both with and without replacement. These functions are exported to R and as a C++ interface and are enabled for use with the default 64 bit generator from the PCG family, Xoroshiro128+/+/\*\* and Xoshiro256+/+/\*\* as well as the 64 bit version of the 20 rounds Threefry engine (Salmon et al., 2011, [doi:10.1145/2063384.2063405](https://doi.org/10.1145/2063384.2063405)) as provided by the package ‘sitmo’.

Author(s)

**Maintainer:** Ralf Stubner <ralf.stubner@gmail.com> ([ORCID](#))

Other contributors:

- daqana GmbH [copyright holder]
- David Blackman (Xoroshiro / Xoshiro family) [copyright holder]
- Melissa O’Neill <oneill@pcg-random.org> (PCG family) [copyright holder]
- Sebastiano Vigna <vigna@acm.org> (Xoroshiro / Xoshiro family) [copyright holder]
- Aaron Lun [contributor]
- Kyle Butts <kyle.butts@colorado.edu> [contributor]
- Henrik Sloot [contributor]
- Philippe Grosjean ([ORCID](#)) [contributor]

**See Also**

Useful links:

- <https://daqana.github.io/dqrng/>
- <https://github.com/daqana/dqrng>
- Report bugs at <https://github.com/daqana/dqrng/issues>

---

dqrnorm

*Multivariate Distributions*

---

**Description**

Multivariate Distributions

**Usage**

```
dqrnorm(n, ...)
```

**Arguments**

|     |                                      |
|-----|--------------------------------------|
| n   | number of observations               |
| ... | forwarded to <a href="#">rmvnorm</a> |

**Value**

numeric matrix of multivariate normal distributed variables

**See Also**

[rmvnorm](#)

**Examples**

```
sigma <- matrix(c(4,2,2,3), ncol=2)
x <- dqrnorm(n=500, mean=c(1,2), sigma=sigma)
colMeans(x)
var(x)
plot(x)
```

dqRNGkind

*R interface***Description**

The dqrng package provides several fast random number generators together with fast functions for generating random numbers according to a uniform, normal and exponential distribution. These functions are modeled after the base functions [set.seed](#), [RNGkind](#), [runif](#), [rnorm](#), and [rexp](#). However, note that the functions provided here do not accept vector arguments for the number of observations as well as the parameters describing the distribution functions. Please see [register\\_methods](#) if you need this functionality.

dqrrademacher uses a fast algorithm to generate random Rademacher variables (-1 and 1 with equal probability). To do so, it generates a random 64 bit integer and then uses each bit to generate a 0/1 variable. This generates 64 integers per random number generation.

dqrng\_get\_state and dqrng\_set\_state can be used to get and set the RNG's internal state. The character vector should not be manipulated directly.

**Usage**

```
dqRNGkind(kind, normal_kind = "ignored")

dqrng_get_state()

dqrng_set_state(state)

dqrnif(n, min = 0, max = 1)

dqrnorm(n, mean = 0, sd = 1)

dqrexp(n, rate = 1)

dqrrademacher(n)

dqset.seed(seed, stream = NULL)
```

**Arguments**

|             |                                                                  |
|-------------|------------------------------------------------------------------|
| kind        | string specifying the RNG (see details)                          |
| normal_kind | ignored; included for compatibility with <a href="#">RNGkind</a> |
| state       | character vector representation of the RNG's internal state      |
| n           | number of observations                                           |
| min         | lower limit of the uniform distribution                          |
| max         | upper limit of the uniform distribution                          |
| mean        | mean value of the normal distribution                            |

|        |                                                                                                                                           |
|--------|-------------------------------------------------------------------------------------------------------------------------------------------|
| sd     | standard deviation of the normal distribution                                                                                             |
| rate   | rate of the exponential distribution                                                                                                      |
| seed   | integer scalar to seed the random number generator, or an integer vector of length 2 representing a 64-bit seed. Maybe NULL, see details. |
| stream | integer used for selecting the RNG stream; either a scalar or a vector of length 2                                                        |

## Details

Supported RNG kinds:

**pcg64** The default 64 bit variant from the PCG family developed by Melissa O'Neill. See <https://www.pcg-random.org/> for more details.

**Xoroshiro128++ and Xoroshiro256++** RNGs developed by David Blackman and Sebastiano Vigna. See <https://prng.di.unimi.it/> for more details. The older generators Xoroshiro128+ and Xoroshiro256+ should be used only for backwards compatibility.

**Threefry** The 64 bit version of the 20 rounds Threefry engine as provided by [sitmo-package](#)

Xoroshiro128++ is the default since it is fast, small and has good statistical properties.

The functions dqnorm and dqexp use the Ziggurat algorithm as provided by `boost.random`.

See [generateSeedVectors](#) for rapid generation of integer-vector seeds that provide 64 bits of entropy. These allow full exploration of the state space of the 64-bit RNGs provided in this package.

If the provided seed is NULL, a seed is generated from R's RNG without state alteration.

## Value

dqrunif, dqnorm, and dqexp return a numeric vector of length `n`. dqrrademacher returns an integer vector of length `n`. dqrng\_get\_state returns a character vector representation of the RNG's internal state.

## See Also

[set.seed](#), [RNGkind](#), [runif](#), [rnorm](#), and [rexp](#)

## Examples

```
library(dqrng)

# Set custom RNG.
dqRNGkind("Xoroshiro256++")

# Use an integer scalar to set a seed.
dqset.seed(42)

# Use integer scalars to set a seed and the stream.
dqset.seed(42, 123)

# Use an integer vector to set a seed.
dqset.seed(c(31311L, 24123423L))
```

```
# Use an integer vector to set a seed and a scalar to select the stream.
dqset.seed(c(31311L, 24123423L), 123)

# Random sampling from distributions.
dqrnif(5, min = 2, max = 10)
dqrexp(5, rate = 4)
dqrnorm(5, mean = 5, sd = 3)

# get and restore the state
(state <- dqrng_get_state())
dqrnif(5)
dqrng_set_state(state)
dqrnif(5)
```

---

dqsample

*Unbiased Random Samples and Permutations*


---

## Description

Unbiased Random Samples and Permutations

## Usage

```
dqsample(x, size, replace = FALSE, prob = NULL)
```

```
dqsample.int(n, size = n, replace = FALSE, prob = NULL)
```

## Arguments

|         |                                                                                         |
|---------|-----------------------------------------------------------------------------------------|
| x       | either a vector of one or more elements from which to choose, or a positive integer.    |
| size    | a non-negative integer giving the number of items to choose.                            |
| replace | should sampling be with replacement?                                                    |
| prob    | a vector of probability weights for obtaining the elements of the vector being sampled. |
| n       | a positive number, the number of items to choose from.                                  |

## See Also

`vignette("sample", package = "dqrng")`, [sample](#) and [sample.int](#)

---

|                     |                                          |
|---------------------|------------------------------------------|
| generateSeedVectors | <i>Generate seed as a integer vector</i> |
|---------------------|------------------------------------------|

---

**Description**

Generate seed as a integer vector

**Usage**

```
generateSeedVectors(nseeds, nwords = 2L)
```

**Arguments**

|        |                                                       |
|--------|-------------------------------------------------------|
| nseeds | Integer scalar, number of seeds to generate.          |
| nwords | Integer scalar, number of words to generate per seed. |

**Details**

Each seed is encoded as an integer vector with the most significant bits at the start of the vector. Each integer vector is converted into an unsigned integer (in C++ or otherwise) by the following procedure:

1. Start with a sum of zero.
2. Add the first value of the vector.
3. Left-shift the sum by 32.
4. Add the next value of the vector, and repeat.

The aim is to facilitate R-level generation of seeds with sufficient randomness to cover the entire state space of pseudo-random number generators that require more than the ~32 bits available in an int. It also preserves the integer nature of the seed, thus avoiding problems with casting double-precision numbers to integers.

It is possible for the seed vector to contain NA\_integer\_ values. This should not be cause for alarm, as R uses -INT\_MAX to encode missing values in integer vectors.

**Value**

A list of length n, where each element is an integer vector that contains nwords words (i.e., 32\*nwords bits) of randomness.

**Author(s)**

Aaron Lun

**Examples**

```
generateSeedVectors(10, 2)
```

```
generateSeedVectors(5, 4)
```

---

|                  |                                         |
|------------------|-----------------------------------------|
| register_methods | <i>Registering as user-supplied RNG</i> |
|------------------|-----------------------------------------|

---

## Description

The random-number generators (RNG) from this package can be registered as user-supplied RNG. This way all `r<dist>` functions make use of the provided fast RNGs.

## Usage

```
register_methods(kind = c("both", "rng"))

restore_methods()
```

## Arguments

`kind` Which methods should be registered? Either "both" or "rng".

## Details

Caveats:

- While `runif` and `dqrnif` as well as `rnorm` and `dqrnorm` will produce the same results, this is not the case for `rexp` and `dqrexp`.
- The `dqr<dist>` functions are still faster than `r<dist>` when many random numbers are generated.
- You can use only the RNG from this package using `register_method("rng")` or both the RNG and the Ziggurat method for normal draws with `register_method("both")`. The latter approach is used by default. Using only the Ziggurat method will give *undefined* behavior and is not supported!
- Calling `dqset.seed(NULL)` re-initializes the RNG from R's RNG. This no longer makes sense when the RNG has been registered as user-supplied RNG. In that case `set.seed(NULL)` needs to be used.
- With R's in-build RNGs one can get access to the internal state using `.Random.seed`. This is not possible here, since the internal state is a private member of the used C++ classes.

You can automatically register these methods when loading this package by setting the option `dqrng.register_methods` to `TRUE`, e.g. with `options(dqrng.register_methods=TRUE)`.

Notes on seeding:

- When a user-supplied RNG is registered, it is also seeded from the previously used RNG. You will therefore get reproducible (but different) whether you call `set.seed()` before or after `register_methods()`.
- When called with a single integer as argument, both `set.seed()` and `dqset.seed()` have the same effect. However, `dqset.seed()` allows you to call it with two integers thereby supplying 64 bits of initial state instead of just 32 bits.

**Value**

Invisibly returns a three-element character vector of the RNG, normal and sample kinds *before* the call.

**See Also**

[RNGkind](#) and [Random.user](#)

**Examples**

```
register_methods()
# set.seed and dqset.seed influence both (dq)runif and (dq)rnorm
set.seed(4711); runif(5)
set.seed(4711); dqrunif(5)
dqset.seed(4711); rnorm(5)
dqset.seed(4711); dqnorm(5)
# similarly for other r<dist> functions
set.seed(4711); rt(5, 10)
dqset.seed(4711); rt(5, 10)
# but (dq)rexp give different results
set.seed(4711); rexp(5, 10)
set.seed(4711); dqrexp(5, 10)
restore_methods()
```

# Index

dqrexp (dqRNGkind), 4  
dqrnorm, 3  
dqrng (dqrng-package), 2  
dqrng-package, 2  
dqrng\_get\_state (dqRNGkind), 4  
dqrng\_set\_state (dqRNGkind), 4  
dqRNGkind, 4  
dqrnorm (dqRNGkind), 4  
dqrrademacher (dqRNGkind), 4  
dqrnorm (dqRNGkind), 4  
dqsample, 6  
dqset.seed (dqRNGkind), 4  
  
generateSeedVectors, 5, 7  
  
Random.user, 9  
register\_methods, 4, 8  
restore\_methods (register\_methods), 8  
rexp, 4, 5  
rmvnorm, 3  
RNGkind, 4, 5, 9  
rnorm, 4, 5  
runif, 4, 5  
  
sample, 6  
sample.int, 6  
set.seed, 4, 5