

# Package ‘randompack’

May 9, 2026

**Type** Package

**Title** Fast Random Number Generation with Multiple Engines and Distributions

**Version** 0.1.3

**Author** Kristján Jónasson [aut, cre]

**Maintainer** Kristján Jónasson <jonasson@hi.is>

**Description** Random number generation library implemented in C with multiple engines and distribution functions, providing an R interface focused on correctness, speed, and reproducibility. Supports various PRNGs including xoshiro256++/\*\*, PCG64, Philox, and ChaCha20, with methods for continuous, discrete, and multivariate distributions.

**License** BSD\_3\_clause + file LICENSE

**Encoding** UTF-8

**Depends** R (>= 4.0.0)

**Imports** R6

**Suggests** testthat (>= 3.0.0)

**Config/testthat/edition** 3

**RoxygenNote** 7.3.2

**URL** <https://github.com/jonasson2/randompack>

**BugReports** <https://github.com/jonasson2/randompack/issues>

**Collate** 'configure.R' 'continuous.R' 'discrete.R'  
'randompack-package.R' 'rng.R'

**NeedsCompilation** yes

**Repository** CRAN

**Date/Publication** 2026-04-22 13:40:02 UTC

## Contents

randompack-package	2
randompack_engines	2
randompack_rng	3

**Index**

7

---

randompack-package	<i>randompack: Fast Random Number Generation with Multiple Engines and Distributions</i>
--------------------	--

---

**Description**

Random number generation library implemented in C with multiple engines and distribution functions, providing an R interface focused on correctness, speed, and reproducibility. Supports various PRNGs including xoshiro256++/\*\*, PCG64, Philox, and ChaCha20, with methods for continuous, discrete, and multivariate distributions.

**Quick Start**

Create an RNG instance with `randompack_rng()` and use its methods to generate random variates:

```
rng <- randompack_rng()           # Default: x256++simd
rng <- randompack_rng("pcg64")    # Specify engine
x <- rng$normal(100, mu=1, sigma=2) # Generate 100 N(1,2) variates
```

**Available Engines**

Among the supported underlying random number generators (engines) are xoshiro256++, xoshiro256\*\*, PCG64 DXSM, sfc64, Philox-4x64, and ChaCha20. See [randompack\\_rng](#) for a complete list.

**Author(s)**

**Maintainer:** Kristján Jónasson <jonasson@hi.is>

**See Also**

[randompack\\_rng](#) for creating and using random number generators; [randompack\\_engines](#) for a list of available engines

---

randompack_engines	<i>Available RNG Engines</i>
--------------------	------------------------------

---

**Description**

Returns a data frame of supported random number generator engines with their descriptions.

**Usage**

```
randompack_engines()
```

**Value**

A data.frame with columns `engine` (short name) and `description` (full name and citation).

---

randpack\_rng                      *Create and Use Random Number Generators*

---

### Description

To create a random number generator (RNG) object use `randpack_rng()`, and to specify the underlying RNG engine use `randpack_rng(engine)` where `engine` is a character string naming the engine (see Available Engines below).

### Usage

```
randpack_rng(engine = "x256++simd", bitexact = FALSE, full_mantissa = FALSE)
```

### Arguments

<code>engine</code>	RNG engine
<code>bitexact</code>	Logical; set TRUE to make samples bit-identical across platforms
<code>full_mantissa</code>	Logical; set TRUE to use full mantissas for floating draws

### Details

Once created, the RNG object provides methods for drawing samples from various distributions (e.g., `$normal()`, `$uniform()`, `$int()`). The object can be configured using configure methods (e.g., `$seed()`, `$randomize()`). Multiple independent RNG objects can be used for parallel random number generation across different processes or threads.

### Value

An RNG object with methods for drawing random variates.

### Available Engines

<code>x256++simd</code>	xorshift256++ with streams (default)
<code>sfc64simd</code>	sfc64 with streams
<code>x256++</code>	xoshiro256++ (Vigna and Blackman, 2018)
<code>x256**</code>	xoshiro256** (Vigna and Blackman, 2018)
<code>x128+</code>	xorshift128+ (Vigna, 2014)
<code>xoro++</code>	xoroshiro128++ (Vigna and Blackman, 2016)
<code>pcg64</code>	PCG64 DXSM (O'Neill, 2014)
<code>sfc64</code>	sfc64 (Chris Doty-Humphrey, 2013)
<code>squares</code>	squares64 (Widynski, 2021)
<code>philox</code>	Philox-4x64 (Salmon and Moraes, 2011)
<code>cwg128</code>	cwg128 (Działa, 2022)
<code>ranlux++</code>	ranlux++ (Sibidanov, 2017)
<code>chacha20</code>	ChaCha20 (Bernstein, 2008)

### Continuous distributions

The RNG object provides methods for generating random variates from common continuous probability distributions. All methods return a numeric vector of length `len`.

`rng$unif(len)` Uniform variates on  $[0,1)$ .

`rng$unif(len, a, b)` Uniform variates on  $[a,b)$  with  $a < b$ .

`rng$normal(len)` Standard normal variates (mean 0 and standard deviation 1).

`rng$normal(len, mu, sigma)` Normal variates with mean `mu` and standard deviation `sigma` (defaults 0 and 1).

`rng$skew_normal(len, mu, sigma, alpha)` Skew-normal variates with location `mu` and scale `sigma` (defaults 0 and 1), and shape `alpha` (required).

`rng$lognormal(len, mu, sigma)` Lognormal variates derived from an underlying normal distribution (defaults 0 and 1).

`rng$exp(len)` Standard exponential variates (scale 1).

`rng$exp(len, scale)` Exponential variates with scale `scale`.

`rng$gamma(len, shape, scale)` Gamma variates with given shape and scale (default 1).

`rng$chi2(len, nu)` Chi-square variates with `nu` degrees of freedom.

`rng$beta(len, a, b)` Beta variates with shape parameters `a` and `b`.

`rng$t(len, nu)` Student's *t* variates with `nu` degrees of freedom.

`rng$f(len, nu1, nu2)` F variates with `nu1` and `nu2` degrees of freedom.

`rng$gumbel(len, mu, beta)` Gumbel variates with location `mu` and scale `beta` (defaults 0 and 1).

`rng$pareto(len, xm, alpha)` Pareto variates with minimum value `xm` and shape `alpha`.

`rng$weibull(len, shape, scale)` Weibull variates with given shape and scale (default 1).

`rng$mvn(n, Sigma, mu = NULL)` Multivariate normal variates as an `n` by `d` matrix, where `d` is the dimension of `Sigma`.

### Discrete distributions

The RNG object provides methods for generating random variates from common discrete distributions and combinatorial constructions.

`rng$int(len, min, max)` Uniform integers on  $[\text{min}, \text{max}]$ .

`rng$perm(n)` Random permutation of  $1:n$ .

`rng$sample(n, k)` Sample `k` elements without replacement from  $1:n$ .

`rng$raw(len)` Generate `len` random bytes as a raw vector.

### Configuration and Copying

Methods for creating, configuring, and managing RNG state. All state-setting methods accept numeric vectors (double or integer) whose elements must be nonnegative whole numbers not exceeding  $2^{32} - 1$ . Where applicable, shorter vectors are padded with zeros.

`rng$seed(seed, spawn_key = integer(0))` Reinitialize the RNG deterministically from `seed` and an optional numeric vector `spawn_key`.

`rng$randomize()` Randomize the RNG state from system entropy.  
`rng$jump(p)` Jump an xor-family or `ranlux++` engine ahead by  $2^p$  steps. The `x128+` and `xoro128++` engines support  $p = 32, 64, 96$ , while `x256++`, `x256**`, `x256++simd`, and `ranlux++` also support  $p = 128$  and  $p = 192$ .  
`rng$advance(delta)` Advance the `pcg64` engine by an arbitrary 128-bit delta. `delta` may be a scalar, interpreted as `[delta, 0]`, or have length up to 4; shorter vectors are zero-padded.  
`rng$duplicate()` Duplicate the RNG, preserving its state.  
`rng$serialize()` Serialize the current RNG state as a raw vector.  
`rng$deserialize(raw_state)` Restore state from a raw vector created by `serialize()`.  
`rng$set_state(state)` Set the engine state directly (advanced use).  
`rng$pcg64_set_inc(inc)` Set the increment of the PCG64 engine. The increment may have length up to 4 and shorter vectors are zero-padded.  
`rng$cwg128_set_weyl(weyl)` Set the Weyl increment of the `cwg128` engine. `weyl` may have length up to 4 and shorter vectors are zero-padded.  
`rng$sfc64_set_abc(abc)` Set the a, b, c state words of the `sfc64` engine. `abc` may have length up to 6 and shorter vectors are zero-padded.  
`rng$chacha_set_nonce(nonce)` Set the nonce of the ChaCha20 engine. The nonce may have length up to 3 and shorter vectors are zero-padded.  
`rng$philox_set_key(key)` Set the key of the Philox engine. The key may have length up to 4 and shorter vectors are zero-padded.  
`rng$squares_set_key(key)` Set the key of the Squares engine. The key may have length up to 2 and shorter vectors are zero-padded.

### See Also

[randompack\\_engines](#) to list all available engines

### Examples

```

# Create an RNG
rng <- randompack_rng()                # Default engine (xoshiro256++)
rng_pcg <- randompack_rng("pcg64")    # Specify engine
rng_chacha <- randompack_rng("chacha20")

# Continuous distributions
x <- rng$unif(5)
x <- rng$normal(100)                   # Standard normal
x <- rng$normal(100, 1, 2)             # N(1,2)
x <- rng$skew_normal(100, mu=0, sigma=1, alpha=2)
x <- rng$lognormal(5)
x <- rng$beta(5, a=2, b=3)
Sigma <- diag(2)
x <- rng$mvn(10, Sigma, mu=c(1,2))

# Discrete distributions
x <- rng$int(5, min=1, max=10)
x <- rng$perm(5)

```

```
x <- rng$sample(10, k=3)
x <- rng$raw(4)

# Configuration and copying
rng$seed(12345) # seed for reproducibility
rng$randomize() # randomize from system entropy
rngm <- randompack_rng(full_mantissa = TRUE) # 53-bit mantissas for doubles
rng2 <- rng$duplicate() # duplicate with same state
identical(rng$unif(3), rng2$unif(3)) # TRUE
raw_state <- rng$serialize() # save state
rng3 <- randompack_rng() # another default RNG
rng3$deserialize(raw_state) # restore state
identical(rng$unif(3), rng3$unif(3)) # TRUE
rng_sq <- randompack_rng("squares") # engine-specific state setting
rng_sq$set_state(c(2, 0, 0, 0)) # counter = (2,0)
rng_sq$squares_set_key(c(3,4))
```

# Index

`randompack` (`randompack-package`), [2](#)  
`randompack-package`, [2](#)  
`randompack_engines`, [2](#), [2](#), [5](#)  
`randompack_rng`, [2](#), [3](#)