

# Package ‘future.apply’

January 7, 2020

**Version** 1.4.0

**Title** Apply Function to Elements in Parallel using Futures

**Depends** R (>= 3.2.0), future (>= 1.15.1)

**Imports** globals (>= 0.12.5)

**Suggests** datasets, stats, tools, listenv (>= 0.8.0), R.rsp, markdown

**VignetteBuilder** R.rsp

**Description** Implementations of `apply()`, `by()`, `eapply()`, `lapply()`, `Map()`, `mapply()`, `replicate()`, `sapply()`, `tapply()`, and `vapply()` that can be resolved using any future-supported backend, e.g. parallel on the local machine or distributed on a compute cluster. These `future_*apply()` functions come with the same pros and cons as the corresponding base-R `*apply()` functions but with the additional feature of being able to be processed via the future framework.

**License** GPL (>= 2)

**LazyLoad** TRUE

**URL** <https://github.com/HenrikBengtsson/future.apply>

**BugReports** <https://github.com/HenrikBengtsson/future.apply/issues>

**RoxygenNote** 7.0.2

**NeedsCompilation** no

**Author** Henrik Bengtsson [aut, cre, cph],  
R Core Team [cph, ctb]

**Maintainer** Henrik Bengtsson <henrikb@braju.com>

**Repository** CRAN

**Date/Publication** 2020-01-07 21:50:02 UTC

## R topics documented:

future.apply . . . . .	2
future_apply . . . . .	3
future_by . . . . .	5
future_eapply . . . . .	6
future_Map . . . . .	11

<b>Index</b>	<b>15</b>
--------------	-----------

future.apply

*future.apply: Apply Function to Elements in Parallel using Futures*

---

## Description

The **future.apply** packages provides parallel implementations of common "apply" functions provided by base R. The parallel processing is performed via the **future** ecosystem, which provides a large number of parallel backends, e.g. on the local machine, a remote cluster, and a high-performance compute cluster.

## Details

Currently implemented functions are:

- `future_apply()`: a parallel version of `apply()`
- `future_by()`: a parallel version of `by()`
- `future_eapply()`: a parallel version of `eapply()`
- `future_lapply()`: a parallel version of `lapply()`
- `future_mapply()`: a parallel version of `mapply()`
- `future_sapply()`: a parallel version of `sapply()`
- `future_tapply()`: a parallel version of `tapply()`
- `future_vapply()`: a parallel version of `vapply()`
- `future_Map()`: a parallel version of `Map()`
- `future_replicate()`: a parallel version of `replicate()`

Reproducibility is part of the core design, which means that perfect, parallel random number generation (RNG) is supported regardless of the amount of chunking, type of load balancing, and future backend being used.

Since these `future_*`() functions have the same arguments as the corresponding base R function, start using them is often as simple as renaming the function in the code. For example, after attaching the package:

```
library(future.apply)
```

code such as:

```
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE,FALSE,FALSE,TRUE))
y <- lapply(x, quantile, probs = 1:3/4)
```

can be updated to:

```
y <- future_lapply(x, quantile, probs = 1:3/4)
```

The default settings in the **future** framework is to process code *sequentially*. To run the above in parallel on the local machine (on any operating system), use:

```
plan(multiprocess)
```

first. That's it!

To go back to sequential processing, use `plan(sequential)`. If you have access to multiple machines on your local network, use:

```
plan(cluster, workers = c("n1", "n2", "n2", "n3"))
```

This will set up four workers, one on n1 and n3, and two on n2. If you have SSH access to some remote machines, use:

```
plan(cluster, workers = c("m1.myserver.org", "m2.myserver.org"))
```

See the **future** package and `future::plan()` for more examples.

The **future.batchtools** package provides support for high-performance compute (HPC) cluster schedulers such as SGE, Slurm, and TORQUE / PBS. For example,

- `plan(batchtools_slurm)`: Process via a Slurm scheduler job queue.
- `plan(batchtools_torque)`: Process via a TORQUE / PBS scheduler job queue.

This builds on top of the queuing framework that the **batchtools** package provides. For more details on backend configuration, please see the **future.batchtools** and **batchtools** packages.

These are just a few examples of parallel/distributed backend for the future ecosystem. For more alternatives, see the 'Reverse dependencies' section on the [future CRAN package page](#).

## Author(s)

Henrik Bengtsson, except for the implementations of `future_apply()`, `future_Map()`, `future_replicate()`, `future_sapply()`, and `future_tapply()`, which are adopted from the source code of the corresponding base R functions, which are licensed under GPL ( $\geq 2$ ) with 'The R Core Team' as the copyright holder. Because of these dependencies, the license of this package is GPL ( $\geq 2$ ).

---

future\_apply

*Apply Functions Over Array Margins via Futures*

---

## Description

`future_apply()` implements `base::apply()` using future with perfect replication of results, regardless of future backend used. It returns a vector or array or list of values obtained by applying a function to margins of an array or matrix.

**Usage**

```
future_apply(
  X,
  MARGIN,
  FUN,
  ...,
  future.globals = TRUE,
  future.packages = NULL,
  future.label = "future_apply-%d"
)
```

**Arguments**

X	an array, including a matrix.
MARGIN	A vector giving the subscripts which the function will be applied over. For example, for a matrix 1 indicates rows, 2 indicates columns, c(1,2) indicates rows and columns. Where X has named dimnames, it can be a character vector selecting dimension names.
FUN	A function taking at least one argument.
future.globals	A logical, a character vector, or a named list for controlling how globals are handled. For details, see below section.
future.packages	(optional) a character vector specifying packages to be attached in the R environment evaluating the future.
future.label	If a character string, then each future is assigned a label <code>sprintf(future.label, chunk_idx)</code> . If TRUE, then the same as <code>future.label = "future_apply-%d"</code> . If FALSE, no labels are assigned.
...	(optional) Additional arguments passed to <code>FUN()</code> , except <code>future.*</code> arguments, which are passed on to <code>future_lapply()</code> used internally.

**Value**

Returns a vector or array or list of values obtained by applying a function to margins of an array or matrix. See [base::apply\(\)](#) for details.

**Author(s)**

The implementations of `future_apply()` is adopted from the source code of the corresponding base R function, which is licensed under GPL ( $\geq 2$ ) with 'The R Core Team' as the copyright holder.

**Examples**

```
## -----
## apply()
## -----
X <- matrix(c(1:4, 1, 6:8), nrow = 2L)
```

```

Y0 <- apply(X, MARGIN = 1L, FUN = table)
Y1 <- future_apply(X, MARGIN = 1L, FUN = table)
print(Y1)
stopifnot(all.equal(Y1, Y0, check.attributes = FALSE)) ## FIXME

Y0 <- apply(X, MARGIN = 1L, FUN = stats::quantile)
Y1 <- future_apply(X, MARGIN = 1L, FUN = stats::quantile)
print(Y1)
stopifnot(all.equal(Y1, Y0))

## -----
## Parallel Random Number Generation
## -----

## Regardless of the future plan, the number of workers, and
## where they are, the random numbers produced are identical

X <- matrix(c(1:4, 1, 6:8), nrow = 2L)

plan(multiprocess)
Y1 <- future_apply(X, MARGIN = 1L, FUN = sample, future.seed = 0xBEEF)
print(Y1)

plan(sequential)
Y2 <- future_apply(X, MARGIN = 1L, FUN = sample, future.seed = 0xBEEF)
print(Y2)

stopifnot(all.equal(Y1, Y2))

```

---

future\_by

*Apply a Function to a Data Frame Split by Factors via Futures*


---

## Description

Apply a Function to a Data Frame Split by Factors via Futures

## Usage

```
future_by(data, INDICES, FUN, ..., simplify = TRUE)
```

## Arguments

data	An R object, normally a data frame, possibly a matrix.
INDICES	A factor or a list of factors, each of length nrow(data).
FUN	a function to be applied to (usually data-frame) subsets of data.

simplify            logical: see [base::tapply](#).  
 ...                Additional arguments pass to [future\\_lapply\(\)](#) and then to FUN().

### Details

Internally, data is grouped by INDICES into a list of data subset elements which is then processed by [future\\_lapply\(\)](#). When the groups differ significantly in size, the processing time may differ significantly between the groups. To correct for processing-time imbalances, adjust the amount of chunking via arguments `future.scheduling` and `future.chunk.size`.

### Value

An object of class "by", giving the results for each subset. This is always a list if `simplify` is false, otherwise a list or array (see [base::tapply](#)). See also [base::by\(\)](#) for details.

### Examples

```
## -----
## by()
## -----
library(datasets) ## warpbreaks
library(stats)   ## lm()

y0 <- by(warpbreaks, warpbreaks[, "tension"],
         function(x) lm(breaks ~ wool, data = x))

plan(multiprocess)
y1 <- future_by(warpbreaks, warpbreaks[, "tension"],
               function(x) lm(breaks ~ wool, data = x))

plan(sequential)
y2 <- future_by(warpbreaks, warpbreaks[, "tension"],
               function(x) lm(breaks ~ wool, data = x))
```

---

future\_eapply

*Apply a Function over a List or Vector via Futures*

---

### Description

`future_lapply()` implements [base::lapply\(\)](#) using futures with perfect replication of results, regardless of future backend used. Analogously, this is true for all the other `future_nnn()` functions.

### Usage

```
future_eapply(
  env,
  FUN,
  ...,
```

```
    all.names = FALSE,  
    USE.NAMES = TRUE,  
    future.label = "future_eapply-%d"  
  )  
  
future_lapply(  
  X,  
  FUN,  
  ...,  
  future.stdout = TRUE,  
  future.conditions = NULL,  
  future.globals = TRUE,  
  future.packages = NULL,  
  future.lazy = FALSE,  
  future.seed = FALSE,  
  future.scheduling = 1,  
  future.chunk.size = NULL,  
  future.label = "future_lapply-%d"  
)  
  
future_replicate(  
  n,  
  expr,  
  simplify = "array",  
  future.seed = TRUE,  
  ...,  
  future.label = "future_replicate-%d"  
)  
  
future_sapply(  
  X,  
  FUN,  
  ...,  
  simplify = TRUE,  
  USE.NAMES = TRUE,  
  future.label = "future_sapply-%d"  
)  
  
future_tapply(  
  X,  
  INDEX,  
  FUN = NULL,  
  ...,  
  default = NA,  
  simplify = TRUE,  
  future.label = "future_tapply-%d"  
)
```

```

future_vapply(
  X,
  FUN,
  FUN.VALUE,
  ...,
  USE.NAMES = TRUE,
  future.label = "future_vapply-%d"
)

```

## Arguments

<code>env</code>	An R environment.
<code>FUN</code>	A function taking at least one argument.
<code>all.names</code>	If TRUE, the function will also be applied to variables that start with a period ( <code>.</code> ), otherwise not. See <code>base::eapply()</code> for details.
<code>USE.NAMES</code>	See <code>base::sapply()</code> .
<code>future.label</code>	If a character string, then each future is assigned a label <code>sprintf(future.label, chunk_idx)</code> . If TRUE, then the same as <code>future.label = "future_lapply-%d"</code> . If FALSE, no labels are assigned.
<code>X</code>	A vector-like object to iterate over.
<code>future.stdout</code>	If TRUE (default), then the standard output of the underlying futures is captured, and re-outputted as soon as possible. If FALSE, any output is silenced (by sinking it to the null device as it is outputted). If NA (not recommended), output is <i>not</i> intercepted.
<code>future.conditions</code>	A character string of conditions classes to be captured and relayed. The default is the same as the <code>condition</code> argument of <code>future::Future()</code> . To not intercept conditions, use <code>conditions = character(0L)</code> . Errors are always relayed.
<code>future.globals</code>	A logical, a character vector, or a named list for controlling how globals are handled. For details, see below section.
<code>future.packages</code>	(optional) a character vector specifying packages to be attached in the R environment evaluating the future.
<code>future.lazy</code>	Specifies whether the futures should be resolved lazily or eagerly (default).
<code>future.seed</code>	A logical or an integer (of length one or seven), or a list of <code>length(X)</code> with pre-generated random seeds. For details, see below section.
<code>future.scheduling</code>	Average number of futures ("chunks") per worker. If <code>0.0</code> , then a single future is used to process all elements of <code>X</code> . If <code>1.0</code> or TRUE, then one future per worker is used. If <code>2.0</code> , then each worker will process two futures (if there are enough elements in <code>X</code> ). If <code>Inf</code> or FALSE, then one future per element of <code>X</code> is used. Only used if <code>future.chunk.size</code> is NULL.
<code>future.chunk.size</code>	The average number of elements per future ("chunk"). If <code>Inf</code> , then all elements are processed in a single future. If NULL, then argument <code>future.scheduling</code> is used.

n	The number of replicates.
expr	An R expression to evaluate repeatedly.
simplify	See <code>base::sapply()</code> and <code>base::tapply()</code> , respectively.
INDEX	A list of one or more factors, each of same length as X. The elements are coerced to factors by <code>as.factor()</code> . See also <code>base::tapply()</code> .
default	See <code>base::tapply()</code> .
FUN.VALUE	A template for the required return value from each <code>FUN(X[ii], ...)</code> . Types may be promoted to a higher type within the ordering logical < integer < double < complex, but not demoted. See <code>base::vapply()</code> for details.
...	(optional) Additional arguments passed to <code>FUN()</code> . For <code>future_*apply()</code> functions and <code>replicate()</code> , any <code>future.*</code> arguments part of ... are passed on to <code>future_lapply()</code> used internally.

## Value

A named (unless `USE.NAMES = FALSE`) list. See `base::eapply()` for details.

For `future_lapply()`, a list with same length and names as X. See `base::lapply()` for details.

`future_replicate()` is a wrapper around `future_sapply()` and return simplified object according to the `simplify` argument. See `base::replicate()` for details. Since `future_replicate()` usually involves random number generation (RNG), it uses `future.seed = TRUE` by default in order produce sound random numbers regardless of future backend and number of background workers used.

For `future_sapply()`, a vector with same length and names as X. See `base::sapply()` for details.

`future_tapply()` returns an array with mode "list", unless `simplify = TRUE` (default) and `FUN` returns a scalar, in which case the mode of the array is the same as the returned scalars. See `base::tapply()` for details.

For `future_vapply()`, a vector with same length and names as X. See `base::vapply()` for details.

## Global variables

Argument `future.globals` may be used to control how globals should be handled similarly how the `globals` argument is used with `future()`. Since all function calls use the same set of globals, this function can do any gathering of globals upfront (once), which is more efficient than if it would be done for each future independently. If `TRUE`, `NULL` or `not` is specified (default), then globals are automatically identified and gathered. If a character vector of names is specified, then those globals are gathered. If a named list, then those globals are used as is. In all cases, `FUN` and any ... arguments are automatically passed as globals to each future created as they are always needed.

## Reproducible random number generation (RNG)

Unless `future.seed = FALSE`, this function guarantees to generate the exact same sequence of random numbers *given the same initial seed / RNG state* - this regardless of type of futures, scheduling ("chunking") strategy, and number of workers.

RNG reproducibility is achieved by pregenerating the random seeds for all iterations (over X) by using L'Ecuyer-CMRG RNG streams. In each iteration, these seeds are set before calling

`FUN(X[[i]], ...)`. *Note, for large `length(X)` this may introduce a large overhead.* As input (`future.seed`), a fixed seed (integer) may be given, either as a full L'Ecuyer-CMRG RNG seed (vector of 1+6 integers) or as a seed generating such a full L'Ecuyer-CMRG seed. If `future.seed = TRUE`, then `.Random.seed` is returned if it holds a L'Ecuyer-CMRG RNG seed, otherwise one is created randomly. If `future.seed = NA`, a L'Ecuyer-CMRG RNG seed is randomly created. If none of the function calls `FUN(X[[i]], ...)` uses random number generation, then `future.seed = FALSE` may be used.

In addition to the above, it is possible to specify a pre-generated sequence of RNG seeds as a list such that `length(future.seed) == length(X)` and where each element is an integer seed vector that can be assigned to `.Random.seed`. One approach to generate a set of valid RNG seeds based on fixed initial seed (here 42L) is:

```
seeds <- future_lapply(seq_along(X), FUN = function(x) .Random.seed,
                      future.chunk.size = Inf, future.seed = 42L)
```

**Note that** `as.list(seq_along(X))` **is not a valid set of such** `.Random.seed` **values.**

In all cases but `future.seed = FALSE`, the RNG state of the calling R processes after this function returns is guaranteed to be "forwarded one step" from the RNG state that was before the call and in the same way regardless of `future.seed`, `future.scheduling` and `future.strategy` used. This is done in order to guarantee that an R script calling `future_lapply()` multiple times should be numerically reproducible given the same initial seed.

### Control processing order of elements

Attribute ordering of `future.chunk.size` or `future.scheduling` can be used to control the ordering the elements are iterated over, which only affects the processing order and *not* the order values are returned. This attribute can take the following values:

- index vector - an numeric vector of length `length(X)`
- function - an function taking one argument which is called as `ordering(length(X))` and which much return an index vector of length `length(X)`, e.g. `function(n) rev(seq_len(n))` for reverse ordering.
- "random" - this will randomize the ordering via random index vector `sample.int(length(X))`. For example, `future.scheduling = structure(TRUE, ordering = "random")`. *Note*, when elements are processed out of order, then captured standard output and conditions are also relayed in that order, that is out of order.

### Author(s)

The implementations of `future_replicate()`, `future_sapply()`, and `future_tapply()` are adopted from the source code of the corresponding base R functions, which are licensed under GPL ( $\geq 2$ ) with 'The R Core Team' as the copyright holder.

### Examples

```
## -----
## lapply(), sapply(), tapply()
## -----
x <- list(a = 1:10, beta = exp(-3:3), logic = c(TRUE, FALSE, FALSE, TRUE))
```

```
y0 <- lapply(x, FUN = quantile, probs = 1:3/4)
y1 <- future_lapply(x, FUN = quantile, probs = 1:3/4)
print(y1)
stopifnot(all.equal(y1, y0))

y0 <- sapply(x, FUN = quantile)
y1 <- future_sapply(x, FUN = quantile)
print(y1)
stopifnot(all.equal(y1, y0))

y0 <- vapply(x, FUN = quantile, FUN.VALUE = double(5L))
y1 <- future_vapply(x, FUN = quantile, FUN.VALUE = double(5L))
print(y1)
stopifnot(all.equal(y1, y0))

## -----
## Parallel Random Number Generation
## -----

## Regardless of the future plan, the number of workers, and
## where they are, the random numbers produced are identical

plan(multiprocess)
y1 <- future_lapply(1:5, FUN = rnorm, future.seed = 0xBEEF)
str(y1)

plan(sequential)
y2 <- future_lapply(1:5, FUN = rnorm, future.seed = 0xBEEF)
str(y2)

stopifnot(all.equal(y1, y2))

## -----
## Process chunks of data.frame rows in parallel
## -----

iris <- datasets::iris
chunks <- split(iris, seq(1, nrow(iris), length.out = 3L))
y0 <- lapply(chunks, FUN = function(iris) sum(iris$Sepal.Length))
y0 <- do.call(sum, y0)
y1 <- future_lapply(chunks, FUN = function(iris) sum(iris$Sepal.Length))
y1 <- do.call(sum, y1)
print(y1)
stopifnot(all.equal(y1, y0))
```

---

**Description**

`future_mapply()` implements `base::mapply()` using futures with perfect replication of results, regardless of future backend used. Analogously to `mapply()`, `future_mapply()` is a multivariate version of `future_sapply()`. It applies FUN to the first elements of each ... argument, the second elements, the third elements, and so on. Arguments are recycled if necessary.

**Usage**

```
future_Map(f, ..., future.label = "future_Map-%d")
```

```
future_mapply(
  FUN,
  ...,
  MoreArgs = NULL,
  SIMPLIFY = TRUE,
  USE.NAMES = TRUE,
  future.stdout = TRUE,
  future.conditions = NULL,
  future.globals = TRUE,
  future.packages = NULL,
  future.lazy = FALSE,
  future.seed = FALSE,
  future.scheduling = 1,
  future.chunk.size = NULL,
  future.label = "future_mapply-%d"
)
```

**Arguments**

<code>f</code>	A function of the arity $k$ if <code>future_Map()</code> is called with $k$ arguments.
<code>future.label</code>	If a character string, then each future is assigned a label <code>sprintf(future.label, chunk_idx)</code> . If TRUE, then the same as <code>future.label = "future_lapply-%d"</code> . If FALSE, no labels are assigned.
<code>FUN</code>	A function to apply, found via <code>base::match.fun()</code> .
<code>MoreArgs</code>	A list of other arguments to FUN.
<code>SIMPLIFY</code>	A logical or character string; attempt to reduce the result to a vector, matrix or higher dimensional array; see the <code>simplify</code> argument of <code>base::sapply()</code> .
<code>USE.NAMES</code>	A logical; use names if the first ... argument has names, or if it is a character vector, use that character vector as the names.
<code>future.stdout</code>	If TRUE (default), then the standard output of the underlying futures is captured, and re-outputted as soon as possible. If FALSE, any output is silenced (by sinking it to the null device as it is outputted). If NA (not recommended), output is <i>not</i> intercepted.

<code>future.conditions</code>	A character string of conditions classes to be captured and relayed. The default is the same as the <code>condition</code> argument of <code>future::Future()</code> . To not intercept conditions, use <code>conditions = character(0L)</code> . Errors are always relayed.
<code>future.globals</code>	A logical, a character vector, or a named list for controlling how globals are handled. For details, see <code>future_lapply()</code> .
<code>future.packages</code>	(optional) a character vector specifying packages to be attached in the R environment evaluating the future.
<code>future.lazy</code>	Specifies whether the futures should be resolved lazily or eagerly (default).
<code>future.seed</code>	A logical or an integer (of length one or seven), or a list of <code>max(lengths(list(...)))</code> with pre-generated random seeds. For details, see <code>future_lapply()</code> .
<code>future.scheduling</code>	Average number of futures ("chunks") per worker. If <code>0.0</code> , then a single future is used to process all elements of <code>X</code> . If <code>1.0</code> or <code>TRUE</code> , then one future per worker is used. If <code>2.0</code> , then each worker will process two futures (if there are enough elements in <code>X</code> ). If <code>Inf</code> or <code>FALSE</code> , then one future per element of <code>X</code> is used. Only used if <code>future.chunk.size</code> is <code>NULL</code> .
<code>future.chunk.size</code>	The average number of elements per future ("chunk"). If <code>Inf</code> , then all elements are processed in a single future. If <code>NULL</code> , then argument <code>future.scheduling</code> is used.
<code>...</code>	Arguments to vectorize over (vectors or lists of strictly positive length, or all of zero length).

**Value**

`future_Map()` is a simple wrapper to `future_mapply()` which does not attempt to simplify the result. See `base::Map()` for details.

`future_mapply()` returns a list, or for `SIMPLIFY = TRUE`, a vector, array or list. See `base::mapply()` for details.

**Author(s)**

The implementations of `future_Map()` is adopted from the source code of the corresponding base R function `Map()`, which is licensed under GPL ( $\geq 2$ ) with 'The R Core Team' as the copyright holder.

**Examples**

```
## -----
## mapply()
## -----
y0 <- mapply(rep, 1:4, 4:1)
y1 <- future_mapply(rep, 1:4, 4:1)
stopifnot(identical(y1, y0))

y0 <- mapply(rep, times = 1:4, x = 4:1)
```

```

y1 <- future_mapply(rep, times = 1:4, x = 4:1)
stopifnot(identical(y1, y0))

y0 <- mapply(rep, times = 1:4, MoreArgs = list(x = 42))
y1 <- future_mapply(rep, times = 1:4, MoreArgs = list(x = 42))
stopifnot(identical(y1, y0))

y0 <- mapply(function(x, y) seq_len(x) + y,
             c(a = 1, b = 2, c = 3), # names from first
             c(A = 10, B = 0, C = -10))
y1 <- future_mapply(function(x, y) seq_len(x) + y,
                   c(a = 1, b = 2, c = 3), # names from first
                   c(A = 10, B = 0, C = -10))
stopifnot(identical(y1, y0))

word <- function(C, k) paste(rep.int(C, k), collapse = "")
y0 <- mapply(word, LETTERS[1:6], 6:1, SIMPLIFY = FALSE)
y1 <- future_mapply(word, LETTERS[1:6], 6:1, SIMPLIFY = FALSE)
stopifnot(identical(y1, y0))

## -----
## Parallel Random Number Generation
## -----

## Regardless of the future plan, the number of workers, and
## where they are, the random numbers produced are identical

plan(multiprocess)
y1 <- future_mapply(stats::runif, n = 1:4, max = 2:5,
                   MoreArgs = list(min = 1), future.seed = 0xBEEF)
print(y1)

plan(sequential)
y2 <- future_mapply(stats::runif, n = 1:4, max = 2:5,
                   MoreArgs = list(min = 1), future.seed = 0xBEEF)
print(y2)

stopifnot(all.equal(y1, y2))

```

# Index

## \*Topic **iteration**

- future.apply, 2
- future\_eapply, 6
- future\_Map, 12

## \*Topic **manip**

- future.apply, 2
- future\_eapply, 6
- future\_Map, 12

## \*Topic **programming**

- future.apply, 2
- future\_eapply, 6
- future\_Map, 12

.Random.seed, 10

apply(), 2

base::apply(), 3, 4

base::by(), 6

base::eapply(), 8, 9

base::lapply(), 6, 9

base::Map(), 13

base::mapply(), 12, 13

base::match.fun(), 12

base::replicate(), 9

base::sapply(), 8, 9, 12

base::tapply, 6

base::tapply(), 9

base::vapply(), 9

by(), 2

eapply(), 2

future.apply, 2

future.apply-package (future.apply), 2

future::Future(), 8, 13

future::plan(), 3

future\_apply, 3

future\_apply(), 2

future\_by, 5

future\_by(), 2

future\_eapply, 6

future\_eapply(), 2

future\_lapply (future\_eapply), 6

future\_lapply(), 2, 6, 13

future\_Map, 11

future\_Map(), 2

future\_mapply (future\_Map), 12

future\_mapply(), 2

future\_replicate (future\_eapply), 6

future\_replicate(), 2

future\_sapply (future\_eapply), 6

future\_sapply(), 2

future\_tapply (future\_eapply), 6

future\_tapply(), 2

future\_vapply (future\_eapply), 6

future\_vapply(), 2

lapply(), 2

Map(), 2

mapply(), 2

replicate(), 2

sapply(), 2

tapply(), 2

vapply(), 2