Package 'broadcast'

January 29, 2025

Title Simple Broadcasted and Type-
Consistent Operations for Atomic and Recursive Arrays with Minimal Dependencies

Version 0.0.0.9

Description Implements simple broadcasted binding and binary operations,
for atomic and recursive arrays.
Besides linking to 'Rcpp',
'broadcast' does not depend on, vendor, link to, or otherwise use any external libraries;
'broadcast' was essentially made from scratch and can be installed out-of-the-box.
The broadcasted implementations include, but are not limited to, the following.
1) Relational operations (like `==`, `!=`, `<`, `>`, `<=`, `>=`; can also take into account Machine precision);
2) Arithmetic operations (like `+`, `-`, `*`, `/`, `^`, `pmin()`, `pmax()`, integer modulo);
3) Boolean combiner operations (like `&`, `|`, `xor()`, ``nand");
4) String distance, (in)equality, and concatenation operations;
5) A Broadcasted implementation of `ifelse()`;
6) A Broadcasted apply-like function;
7) Binding arrays along any arbitrary axis, with broadcast support.
The broadcasted implementations strive to minimize computation time and memory usage
(which is not just good for computer efficiency, but also for the environment).

License MPL-2.0 | file LICENSE

Encoding UTF-8

LinkingTo Rcpp

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.1

Depends R (>= 4.2.0)

Imports Rcpp (>= 1.0.11),
data.table

Suggests tinytest, roxygen2, altdoc, kableExtra

Contents

`aaa00_broadcast_help` *broadcast: Simple Broadcasted Operations for Atomic and Recursive Arrays with Minimal Dependencies*

Description

broadcast:
Simple Broadcasted Binding and Binary Operations for Atomic and Recursive Arrays with Minimal Dependencies.

Implements simple broadcasted binding and binary operations, for atomic and recursive arrays.

Besides linking to 'Rcpp', 'broadcast' does not depend on, vendor, link to, or otherwise use any external libraries; 'broadcast' was essentially made from scratch and can be installed out-of-the-box.

The broadcasted implementations include, but are not limited to, the following:

1. Relational operations (like `==`, `!=`, `<`, `>`, `<=`, `>=`; can also take into account Machine precision);
2. Arithmetic operations (like `+`, `-`, `*`, `/`, `^`, `pmin()`, `pmax()`, integer modulo);
3. Boolean combiner operations (like `&`, `|`, `xor()`, "nand");
4. String distance, (in)equality, and concatenation operations;
5. A Broadcasted implementation of `ifelse()`;
6. A Broadcasted apply-like function;
7. Binding arrays along any arbitrary axis, with broadcast support.

The broadcasted implementations strive to minimize computation time and memory usage (which is not just good for computer efficiency, but also for the environment).

Getting Started

An introduction and overview of the package can be found on the website.

Methods and Functions

Type Specific Binary Operations

'broadcast' provides a set of functions for type-specific binary operations for broadcasted operations.
These functions use an API similar to the outer and sweep functions.

The following functions for type-specific binary operations are available:

- bc.num: numeric arithmetic and relational operations;
- bc.b: Boolean combiner operations;
- bc.cplx: complex arithmetic and equality operations;
- bc.str: string equality, concatenation, and distance operations;
- bc.list: apply any 'R' function to 2 recursive arrays with broadcasting.

General functions
'broadcast' also comes with 2 general broadcasted functions:

- bc_ifelse: Broadcasted version of ifelse.
- bcapply: Broadcasted apply-like function.

Binding Implementations
'broadcast' provides 3 binding implementations:
bind_mat, bind_array, and bind_dt.

Other functions
'broadcast' also provides type-casting functions, which preserve names and dimensions - convenient for arrays.

Author(s)

Author, Maintainer: Tony Wilkes <tony_a_wilkes@outlook.com> (ORCID)

References

The badges shown in the documentation of this R-package were made using the services of: https://shields.io/

`array_recycle` *Recycle Array Dimensions*

Description

The `array_recycle()` function recycles array dimensions until the specified dimension sizes are reached, and returns the array.

The various broadcasting functions "recycle" an array virtually, meaning little to no additional memory is needed.
The `array_recycle()` function, however, physically recycles an array (and thus actually occupies memory space).

Usage

```
array_recycle(x, tdim)
```

Arguments

x	an atomic or recursive array or matrix.
tdim	an integer vector, giving the target dimension to reach.

Value

Returns the recycled array.

Examples

```
x <- matrix(1:9, 3,3)
colnames(x) <- LETTERS[1:3]
rownames(x) <- letters[1:3]
names(x) <- month.abb[1:9]
print(x)

array_recycle(x, c(3,3,2)) # recycle to larger size
```

---

bc.b *Broadcasted Operations for Logical Arrays*

Description

The bc.b() function performs broadcasted operations on 2 logical arrays.

Usage

```
bc.b(x, y, op)
```

Arguments

x, y	conformable atomic arrays of types logical, integer, or double.
op	a single string, giving the operator. Supported Boolean combiner operators: &, |, xor, nand. Supported relational operators: ==, !=, <, >, <=, >=.

Value

For the boolean combiner operators:
A logical array as a result of the broadcasted arithmetic operation.

For relational operators:
A logical array as a result of the broadcasted relational comparison.

Examples

```
x.dim <- c(10:8)
x.len <- prod(x.dim)
x.data <- sample(c(TRUE, FALSE, NA), x.len, TRUE)
x <- array(x.data, x.dim)
y <- array(1:50, c(10,1,1))

bc.b(x, y, "&")
bc.b(x, y, "|")
bc.b(x, y, "xor")
bc.b(x, y, "nand")

bc.b(x, y, "==")
bc.b(x, y, "!=")
bc.b(x, y, "<")
bc.b(x, y, ">")
bc.b(x, y, "<=")
bc.b(x, y, ">=")
```

`bc.cplx` *Broadcasted Operations for Complex Number Arrays*

Description

The `bc.cplx()` function performs broadcasted operations on 2 `complex` arrays.

Note that `bc.cplx()` uses more strict `NA` checks than base 'R':
If for an element of either `x` or `y`, either the real or imaginary part is `NA` or `NaN`, than the result of the operation for that element is necessarily `NA`.

Usage

```
bc.cplx(x, y, op)
```

Arguments

`x`, `y`	conformable atomic arrays of typee `complex`.
`op`	a single string, giving the operator. Supported arithmetic operators: +, -, *, /. Supported relational operators: ==, !=.

Value

For arithmetic operators:
A complex array as a result of the broadcasted arithmetic operation.

For relational operators:
A logical array as a result of the broadcasted relational comparison.

Examples

```
x.dim <- c(10:8)
x.len <- prod(x.dim)
gen <- function() sample(c(rnorm(10), NA, NA, NaN, NaN, Inf, Inf, -Inf, -Inf))
x <- array(gen() + gen() * -1i, x.dim)
y <- array(gen() + gen() * -1i, c(10,1,1))

bc.cplx(x, y, "==")
bc.cplx(x, y, "!=")

bc.cplx(x, y, "+")

bc.cplx(array(gen() + gen() * -1i), array(gen() + gen() * -1i), "==")
bc.cplx(array(gen() + gen() * -1i), array(gen() + gen() * -1i), "!=")

x <- gen() + gen() * -1i
y <- gen() + gen() * -1i
out <- bc.cplx(array(x), array(y), "*")
cbind(x, y, x*y, out)
```

---

bc.list *Broadcasted Operations for Recursive Arrays*

Description

The `bc.list()` function performs broadcasted operations on 2 Recursive arrays.

Usage

```
bc.list(x, y, f)
```

Arguments

x, y	conformable Recursive arrays (i.e. arrays of type `list`).
f	a function that takes in exactly **2** arguments, and **returns** a result that can be stored in a single element of a list.

Value

A recursive array.

Examples

```
x.dim <- c(c(10, 2,2))
x.len <- prod(x.dim)

gen <- function(n) sample(list(letters, month.abb, 1:10), n, TRUE)

x <- array(gen(10), x.dim)
y <- array(gen(10), c(10,1,1))
```

```
bc.list(
  x, y,
  \(x, y)c(length(x) == length(y), typeof(x) == typeof(y))
)
```

---

bc.num *Broadcasted Operations for Numeric Arrays*

Description

The bc.num() function performs broadcasted operations on 2 numeric arrays.

Usage

```
bc.num(x, y, op, prec = sqrt(.Machine$double.eps))
```

Arguments

x, y	conformable atomic arrays of types logical, integer, or double.
op	a single string, giving the operator. Supported arithmetic operators: +, -, *, /, ^, pmin, pmax. Supported relational operators: ==, !=, <, >, <=, >=, d==, d!=, d<, d>, d<=, d>=.
prec	a single number between 0 and 0.1, giving the machine precision to use. Only relevant for the following operators: d==, d!=, d<, d>, d<=, d>= See the d==, d!=, d<, d>, d<=, d>= operators from the 'tinycodet' package for details.

Value

For arithmetic operators:
A numeric array as a result of the broadcasted arithmetic operation.

For relational operators:
A logical array as a result of the broadcasted relational comparison.

Examples

```
x.dim <- c(10:8)
x.len <- prod(x.dim)
x.data <- sample(c(NA, 1.1:1000.1), x.len, TRUE)
x <- array(x.data, x.dim)
y <- array(1:50, c(10,1,1))

bc.num(x, y, "+")
bc.num(x, y, "-")
```

```
bc.num(x, y, "*")
bc.num(x, y, "/")
bc.num(x, y, "^")

bc.num(x, y, "==")
bc.num(x, y, "!=")
bc.num(x, y, "<")
bc.num(x, y, ">")
bc.num(x, y, "<=")
bc.num(x, y, ">=")
```

`bc.str` *Broadcasted Operations for Character/String Arrays*

Description

The `bc.str()` function performs broadcasted operations on 2 character/string arrays.
`bc.chr()` is an alias for `bc.str()`.

Usage

```
bc.str(x, y, op)

bc.chr(x, y, op)
```

Arguments

`x, y`	conformable atomic arrays of typee `character`.
`op`	a single string, giving the operator. Supported concatenation operators: +. Supported relational operators: ==, !=. Supported distance operators: levenshtein.

Value

For concatenation operation:
A character array as a result of the broadcasted concatenation operation.

For relational operation:
A logical array as a result of the broadcasted relational comparison.

For distance operation:
An integer array as a result of the broadcasted relational comparison.

Examples

```
# string concatenation:
x <- array(letters, c(10, 2, 1))
y <- array(letters, c(10,1,1))
bc.str(x, y, "+")


# string (in)equality:
bc.str(array(letters), array(letters), "==")
bc.str(array(letters), array(letters), "!=")


# string distance (Levenshtein):
x <- array(month.name, c(12, 1))
y <- array(month.abb, c(1, 12))
out <- bc.str(x, y, "levenshtein")
dimnames(out) <- list(month.name, month.abb)
print(out)
```

---

bcapply *Apply a Function to 2 Broadcasted Arrays*

Description

The `bcapply()` function applies a function to 2 arrays with broadcasting.

Usage

```
bcapply(x, y, f, v = "list")
```

Arguments

`x`, `y`	conformable atomic or recursive arrays.
`f`	a function that takes in exactly **2** arguments, and **returns** a result that can be stored in a single element of a recursive or atomic array.
`v`	a single string, giving the scalar type for a single iteration. If `NULL` or `"list"` (default), the result will be a recursive array. If it is certain that, for every iteration, `f()` always results in a **single atomic scalar**, the user can specify the type in `v` to pre-allocate the result. Pre-allocating the results leads to slightly faster and more memory efficient code. NOTE: Incorrectly specifying `v` leads to undefined behaviour.

Value

An atomic or recursive array with dimensions `bc_dim(x, y)`.

Examples

```
x.dim <- c(c(10, 2,2))
x.len <- prod(x.dim)

gen <- function(n) sample(list(letters, month.abb, 1:10), n, TRUE)

x <- array(gen(10), x.dim)
y <- array(gen(10), c(10,1,1))

f <- function(x, y) list(x, y)
bcapply(x, y, f)
```

---

bc_dim *Predict Broadcasted dimensions*

Description

bc_dim(x, y) gives the dimensions an array would have, as the result of an broadcasted binary element-wise operation between 2 arrays x and y.

Usage

```
bc_dim(x, y)
```

Arguments

x, y	an atomic array or matrix.

Value

Returns the recycled array.

Examples

```
x.dim <- c(10:8)
x.len <- prod(x.dim)
x.data <- sample(c(TRUE, FALSE, NA), x.len, TRUE)
x <- array(x.data, x.dim)
y <- array(1:50, c(10,1,1))

dim(bc.b(x, y, "&")) == bc_dim(x, y)
dim(bc.b(x, y, "|")) == bc_dim(x, y)
```

bc_ifelse *Broadcasted Ifelse*

Description

The bc_ifelse() S4 generic method performs a broadcasted form of ifelse.

Usage

```
bc_ifelse(cond, yes, no)
```

Arguments

cond	logical vector or array with the length equal to prod(bc_dim(yes, no)).
yes, no	conformable arrays of the same type. All atomic types (see atomic) are supported. Recursive arrays of type list are also supported. since bc_ifelse() is an S4 generic, it can be extended to support special array classes.

Value

The ouput, here referred to as out, will be an array of the same type as yes and no.
After broadcasting yes against no, given any element index i, the following will hold for the output:

- when cond[i] == TRUE, out[i] is yes[i];
- when cond[i] == FALSE, out[i] is no[i];
- when cond[i] is NA, out[i] is NA when yes and no are atomic, and out[i] is list(NULL) when yes and no are recursive.

Examples

```
x.dim <- c(c(10, 2,2))
x.len <- prod(x.dim)

gen <- function(n) sample(list(letters, month.abb, 1:10), n, TRUE)

x <- array(gen(10), x.dim)
y <- array(gen(10), c(10,1,1))

cond <- bc.list(
  x, y,
  \(x, y)c(length(x) == length(y) && typeof(x) == typeof(y))
) |> as_bool()

bc_ifelse(cond, yes = x, no = y)
```

bind *Dimensional Binding of Objects*

Description

The bind_implementations provide dimensional binding functionalities.

The following implementations are available:

- `bind_mat()` binds dimensionless (atomic/recursive) vectors and (atomic/recursive) matrices row- or column-wise.
  Allows for recycling.
- `bind_array()` binds (atomic/recursive) arrays and (atomic/recursive) matrices.
  Allows for broadcasting.
- `bind_dt()` binds data.tables and other data.frame-like objects.
  This function is only available if the 'data.table' package is installed.
  Returns a `data.table`.
  Faster than `do.call(cbind, ...)` or `do.call(rbind, ...)` for regular `data.frame` objects.

Note that the naming convention of the binding implementations here is "bind_" followed by the **resulting class** (abbreviated).
I.e. `bind_mat` **returns** a matrix, but can bind both matrices and vectors.
And `bind_array` **returns** an array, but can bind both arrays and matrices.
And `bind_dt` **returns** a data.table, but can bind not only data.tables, but also most other data.frame-like objects.

Usage

```
bind_mat(input, along, name_deparse = TRUE, comnames_from = 1L)

bind_array(
  input,
  along,
  max_bc = 1L,
  name_along = TRUE,
  comnames_from = 1L,
  name_flat = FALSE
)

bind_dt(input, along, ...)
```

Arguments

`input`	a list of only the appropriate objects. If `input` is named, its names will be used for the names of dimension `along` of the output, as far as possible.

along — a single integer, indicating the dimension along which to bind the dimensions.
I.e. use along = 1 for row-binding, along = 2 for column-binding, etc.
For arrays, additional flexibility is available:

- Specifying along = 0 will bind the arrays on a new dimension before the first, making along the new first dimension.
- Specifying along = n+1, with n being the last available dimension, will create an additional dimension (n+1) and bind the arrays along that new dimension.

name_deparse — Boolean, for bind_mat().
Indicates if dimension along should be named.
Uses the naming method from rbind/cbind itself.

comnames_from — either integer scalar or NULL, for bind_mat() and bind_array().
Indicates which object in input should be used for naming the shared dimension.
If NULL, no communal names will be given.
For example:
When binding columns of matrices, the matrices will share the same rownames.
Using comnames_from = 10 will then result in bind_array() using rownames(input[[10]]) for the rownames of the output.

max_bc — integer, for bind_array.
Specify here the number of dimensions that are allowed to be broadcasted when binding arrays.
If max_bc = 0L, **no** broadcasting will be allowed at all.

name_along — Boolean, for bind_array().
Indicates if dimension along should be named.

name_flat — Boolean, for bind_array().
Indicates if flat indices should be named.
Note that setting this to TRUE will reduce performance considerably.
for performance: set to FALSE

... — arguments to be passed to rbindlist.

Details

The API of bind_array() is inspired by the fantastic abind::abind function by Tony Plare & Richard Heiberger (2016).
But bind_array() differs considerably from abind::abind in the following ways:

- bind_array() differs from abind::abind in that it can handle recursive arrays properly (the abind::abind function would unlist everything to atomic arrays, ruining the structure).
- bind_array() allows for broadcasting, while abind::abind does not support broadcasting.
- bind_array() is generally faster than abind::abind, as bind_array() relies heavily on 'C' and 'C++' code.
- unlike abind::abind, bind_array() only binds (atomic/recursive) arrays and matrices. bind_array()does not attempt to convert things to arrays when they are not arrays, but will give an error instead.
  This saves computation time and prevents unexpected results.
- bind_array() has more streamlined naming options, compared to abind::abind.

`bind_mat()` is a modified version of rbind/cbind.
The primary differences is that `bind_mat()` gives an error when fractional recycling is attempted (like binding `1:3` with `1:10`).

Value

The bound object.

References

Plate T, Heiberger R (2016). *abind: Combine Multidimensional Arrays*. R package version 1.4-5, https://CRAN.R-project.org/package=abind.

Examples

```
# bind_array ====

# here, atomic and recursive arrays are mixed,
# resulting in recursive arrays

# creating the arrays
x <- c(
  lapply(1:3, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:3, \(x)sample(1:10)),
  lapply(1:3, \(x)rnorm(10)),
  lapply(1:3, \(x)sample(letters))
)
x <- matrix(x, 4, 3, byrow = TRUE)
dimnames(x) <- list(letters[1:4], LETTERS[1:3])
print(x)

y <- matrix(1:12, 4, 3)
print(y)

# binding the arrays
input <- list(x = x, y = y)
bind_array(input, along = 0L) # binds on new dimension before first
bind_array(input, along = 1L) # binds on first dimension
bind_array(input, along = 2L)
bind_array(input, along = 3L) # bind on new dimension after last


# binding, wwith empty arrays
emptyarray <- array(numeric(0L), c(0L, 3L))
dimnames(emptyarray) <- list(NULL, paste("empty", 1:3))
print(emptyarray)
input <- list(x = x, y = emptyarray)
bind_array(input, along = 1L, comnames_from = 2L) # row-bind



################################################################################

# bind_mat ====
```

```
# here, atomic and recursive matrices are mixed,
# resulting in a recursive matrix

x <- c(
  lapply(1:3, \(x)sample(c(TRUE, FALSE, NA))),
  lapply(1:3, \(x)sample(1:10)),
  lapply(1:3, \(x)rnorm(10)),
  lapply(1:3, \(x)sample(letters))
)
x <- matrix(x, 4, 3, byrow = TRUE)
dimnames(x) <- list(letters[1:4], LETTERS[1:3])
print(x)

y <- matrix(1:12, 4, 3)
print(y)

bind_mat(list(x = x, y = y), 2L)



################################################################################

# bind_dt ====


x <- data.frame(a = 1:12, b = month.abb) # data.frame
y <- data.table::data.table(a = 1:12, b = month.abb) # data.table

bind_dt(list(x = x, y = y), 2L) # column bind

bind_dt(list(x = x, y = y), 1L) # row bind
```

`typecast` *Atomic and List Type Casting With Names and Dimensions Preserved*

Description

Type casting usually strips away attributes of objects.
The functions provided here preserve dimensions, dimnames, and names, which may be more convenient for arrays and array-like objects.

The functions are as follows:

- `as_bool()`: converts object to atomic type `logical` (`TRUE`, `FALSE`, `NA`).
- `as_int()`: converts object to atomic type `integer`.
- `as_dbl()`: converts object to atomic type `double` (AKA numeric).
- `as_chr()`: converts object to atomic type `character`.
- `as_cplx()`: converts object to atomic type `complex`.
- `as_raw()`: converts object to atomic type `raw`.

- `as_list()`: converts object to recursive type `list`.

`as_num()` is an alias for `as_dbl()`.
`as_str()` is an alias for `as_chr()`.

See also typeof.

Usage

```
as_bool(x, ...)

as_int(x, ...)

as_dbl(x, ...)

as_num(x, ...)

as_chr(x, ...)

as_str(x, ...)

as_cplx(x, ...)

as_raw(x, ...)

as_list(x, ...)
```

Arguments

`x`	an R object.
`...`	further arguments passed to or from other methods.

Value

The converted object.

Examples

```
# matrix example ====
x <- matrix(sample(-1:28), ncol = 5)
colnames(x) <- month.name[1:5]
rownames(x) <- month.abb[1:6]
names(x) <- c(letters[1:20], LETTERS[1:10])
print(x)

as_bool(x)
as_int(x)
as_dbl(x)
as_chr(x)
```

```
as_cplx(x)
as_raw(x)


################################################################################

# factor example ====
x <- factor(month.abb, levels = month.abb)
names(x) <- month.name
print(x)

as_bool(as_int(x) > 6)
as_int(x)
as_dbl(x)
as_chr(x)
as_cplx(x)
as_raw(x)
```

Index