Package 'broadcast'

September 15, 2025

Title Broadcasted Array Operations Like 'NumPy'

Version 0.1.3

Description Implements efficient 'NumPy'-like broadcasted operations for atomic and recursive arrays.
Besides linking to 'Rcpp',
'broadcast' does not use any external libraries in any way;
'broadcast' was essentially made from scratch and can be installed out-of-the-box.
The implementations available in 'broadcast' include, but are not limited to, the following.
1) Broadcasted element-wise operations on any 2 arrays;
they support a large set of
relational, arithmetic, Boolean, string, and bit-wise operations.
2) A faster, more memory efficient, and broadcasted abind-like function,
for binding arrays along an arbitrary dimension.
3) Broadcasted ifelse-like, and apply-like functions.
4) Casting functions,
that cast subset-
groups of an array to a new dimension, cast nested lists to dimensional lists, and vice-versa.
5) A few linear algebra functions for statistics.
The functions in the 'broadcast' package strive to minimize computation time and memory usage
(which is not just good for efficient computing, but also for the environment).

License MPL-2.0

Encoding UTF-8

LinkingTo Rcpp

RoxygenNote 7.3.2

Depends R (>= 4.2.0)

Imports Rcpp (>= 1.0.14), methods

Suggests tinytest, abind, roxygen2

URL https://github.com/tony-aw/broadcast,
https://tony-aw.github.io/broadcast/

BugReports https://github.com/tony-aw/broadcast/issues/

Language en-gb

NeedsCompilation yes

Author Tony Wilkes [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0001-9498-8379>)

Maintainer Tony Wilkes <tony_a_wilkes@outlook.com>

Repository CRAN

Date/Publication 2025-09-15 09:10:18 UTC

Contents

`aaa00_broadcast_help` *broadcast Package Overview*

Description

broadcast:
Broadcasted Array Operations Like 'NumPy'

Implements efficient 'NumPy'-like broadcasted operations for atomic and recursive arrays.

Besides linking to 'Rcpp', 'broadcast' does not use any external libraries in any way; 'broadcast' was essentially made from scratch and can be installed out-of-the-box.

The implementations available in 'broadcast' include, but are not limited to, the following:

1. Broadcasted element-wise operations on any 2 arrays; they support a large set of relational, arithmetic, Boolean, string, and bit-wise operations.
2. A faster, more memory efficient, and broadcasted abind-like function, for binding arrays along an arbitrary dimension.
3. Broadcasted ifelse-like, and apply-like functions.
4. Casting functions, that cast subset-groups of an array to a new dimension, cast nested lists to dimensional lists, and vice-versa.
5. A few linear algebra functions for statistics.

The functions in the 'broadcast' package strive to minimize computation time and memory usage (which is not just good for efficient computing, but also for the environment).

In the context of operations involving 2 (or more) arrays, "broadcasting" refers to recycling array dimensions without allocating additional memory, which is considerably faster and more memory-efficient than R's regular dimensions replication mechanism.

Links to Get Started

- Online Vignettes: https://tony-aw.github.io/broadcast/vignettes/a_readme.html
- GitHub main page: https://github.com/tony-aw/broadcast
- Reporting Issues or Giving Suggestions: https://github.com/tony-aw/broadcast/issues

Functions

Broadcasted Operators

Base 'R' comes with relational (==, !=, etc.), arithmetic (+, -, *, /, etc.), and logical/bit-wise (&, |) operators.
'broadcast' provides 2 ways to use these operators with broadcasting.

The first (and simple) way is to use the broadcaster class, which comes with it's own method dispatch for the above mentioned operators.

This method support operator precedence, and for the average 'R' user, this is sufficient.

The second way is to use the large set of `bc.-` functions.
These offer much greater control and more operators than the previous method, and has less risk of running into conflicting methods.
But it does not support operator precedence.

More information about both methods can be found here:
broadcast_operators.

Binding Arrays
'broadcast' provides the bind_array function, to bind arrays along an arbitrary dimension, with support for broadcasting.

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()` allows for broadcasting, while `abind::abind` does not support broadcasting.
- `bind_array()` is generally faster and more memory-efficient than `abind::abind`, as `bind_array()` relies heavily on 'C' and 'C++' code.
- `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).
- 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`.

See bind_array.

Casting Functions
'broadcast' provides several "casting" functions.
These can facility complex forms of broadcasting that would normally not be possible.
But these "casting" functions also have their own merit, beside empowering complex broadcasting.

More information about the casting functions can be found here:
broadcast_casting.

General Pairwise Broadcasted Functions
'broadcast' also comes with 2 general pairwise broadcasted functions:

- bc_ifelse: Broadcasted version of ifelse.

- bcapply: Broadcasted apply-like function.

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

'broadcast' also provides simple linear algebra functions for statistics.

And 'broadcast' comes with some helper functions:
bc_dim, ndim, lst.ndim, rep_dim.

Supported Structures

'broadcast' supports atomic/recursive arrays (up to 16 dimensions), and atomic/recursive vectors.
As in standard Linear Algebra Convention, dimensionless vectors are interpreted as column-vectors in broadcasted array operations.

Author(s)

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

References

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

Harris, C.R., Millman, K.J., van der Walt, S.J. et al. *Array programming with NumPy*. Nature 585, 357–362 (2020). doi:10.1038/s4158602026492. (Publisher link).

`aaa01_broadcast_operators`

Details on Broadcasted Operators

Description

Base 'R' comes with relational (==, !=, etc.), arithmetic (+, -, *, /, etc.), and logical/bit-wise (&, |) operators.
'broadcast' provides 2 ways to use these operators with broadcasting.

The first (and simple) way is to use the broadcaster class, which comes with it's own method dispatch for the above mentioned operators.
This approach supports operator precedence, and for the average 'R' user, this is sufficient.

The second way is to use the large set of `bc.-` functions.
These offer much greater control and more operators than the previous method, and has less risk of running into conflicting methods.
But it does not support operator precedence.

Operators Overloaded via Broadcaster Class

The 'broadcast' package provides the broadcaster class, which comes with its own method dispatch for the base operators.
If at least one of the 2 arguments of the base operators has the `broadcaster` class attribute, and no other class (like `bit64`) interferes, broadcasting will be used.

The following arithmetic operators have a 'broadcaster' method: +, -, *, /, ^, %%, %/%
The following relational operators have a 'broadcaster' method: ==, !=, <, >, <=, >=
And finally, the & and | operators also have a 'broadcaster' method.

As the `broadcaster` operator methods simply overload the base operators, operator precedence rules are preserved for the `broadcaster` operator methods.

See also the Examples section below.

Available `bc.` functions

'broadcast' provides a set of functions for broadcasted element-wise binary operations with broadcasting.
These functions use an API similar to the outer function.

The following functions for simple operations are available:

- bc.rel: General relational operations.
- bc.b: Boolean (i.e. logical) operations;
- bc.i: integer arithmetic operations;
- bc.d: decimal arithmetic operations;
- bc.cplx: complex arithmetic operations;
- bc.str: string (in)equality, concatenation, and distance operations;
- bc.raw: operations that take in arrays of type `raw` and return an array of type `raw`;
- bc.bit: BIT-WISE operations, supporting the `raw` and `integer` types;
- bc.list: apply any 'R' function to 2 recursive arrays with broadcasting.

Note that the bc.rel method is the primary method for relational operators (==, !=, <, >, <=, >=), and provides what most user usually need in relational operators.
The various other bc. methods have specialized relational operators available for very specialised needs.

Attribute Handling

The bc. functions and the overloaded operators generally do **not** preserve attributes, unlike the base 'R' operators, except for (dim)names and the broadcaster class attribute (and related attributes).

Broadcasting often results in an object with more dimensions, larger dimensions, and/or larger length than the original objects.
This is relevant as some class-specific attributes are only appropriate for certain dimensions or lengths.
Custom matrix classes, for example, presumes an object to have exactly 2 dimensions.
And the various classes provided by the 'bit' package have length-related attributes.
So class attributes cannot be guaranteed to hold for the resulting objects when broadcasting is involved.

The bc. functions and the overloaded operators **always** preserve the "broadcaster" attribute, as this is necessary to chain together broadcasted operations.

Almost all functions provided by 'broadcast' are S3 or S4 generics;
methods can be written for them for specific classes, so that class-specific attributes can be supported when needed.

Unary operations (i.e. + x, - x) return the original object, with only the sign adjusted.

Examples

```
# maths ====

x <- 1:10
broadcaster(x) <- TRUE

y <- 1:10
broadcaster(y) <- TRUE

x + y / x
x + 1 # mathematically equivalent to the above, since x == y
(x + y) / x
2 * x/x # mathematically equivalent to the above, since x == y

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

```
x + y / x
(x + y) / x

(x + y) * x


# relational operators ====
x <- 1:10
y <- array(1:10, c(1, 10))
broadcaster(x) <- TRUE
broadcaster(y) <- TRUE

x == y
x != y
x < y
x > y
x <= y
x >= y
```

`aaa02_broadcast_casting`

Details on Casting Functions

Description

'broadcast' provides several "casting" functions.
These can facility complex forms of broadcasting that would normally not be possible.
But these "casting" functions also have their own merit, beside empowering complex broadcasting.

The following casting functions are available:

- acast:
  casts group-based subsets of an array into a new dimension.
  Useful for, for example, performing **grouped** broadcasted operations.

- cast_hier2dim:
  casts a nested/hierarchical list into a dimensional list (i.e. array of type `list`).
  Useful because one cannot broadcast through nesting, but one **can** broadcast along dimensions.
- cast_dim2hier:
  casts a dimensional list into a nested/hierarchical list; the opposite of cast_hier2dim.

- cast_dim2flat:
  casts a dimensional list into a flattened list, but with names that indicate their original dimensional positions.
  Mostly useful for printing or summarizing dimensional lists.

- dropnests:
  drop redundant nesting in lists; mostly used for facilitating the above casting functions.

Shared Argument `in2out`

The hier2dim, cast_hier2dim, and cast_dim2hier methods all have the `in2out` argument.

`in2out = TRUE`
By default `in2out` is `TRUE`.
This means the call
`y <- cast_hier2dim(x)`
will cast the elements of the deepest valid depth of `x` to the rows of `y`, and elements of the depth above that to the columns of `y`, and so on until the surface-level elements of `x` are cast to the last dimension of `y`.

Similarly, the call
`x <- cast_dim2hier(y)`
will cast the rows of `y` to the inner most elements of `x`, and the columns of `y` to one depth above that, and so on until the last dimension of `y` is cast to the surface-level elements of `x`.

Consider the nested list `x` with a depth of `3`, and the recursive array `y` with 3 dimensions, where their relationship can described as the following code:
`y <- cast_hier2dim(x)`
`x <- cast_dim2hier(y)`.
Then it holds that:
`x[[i]][[j]][[k]]` corresponds to `y[[k, j, i]]`,
$\forall$(`i`, `j`, `k`) , provided `x[[i]][[j]][[k]]` exists.

`in2out = FALSE`
If `in2out = FALSE`, the call
`y <- cast_hier2dim(x, in2out = FALSE)`
will cast the surface-level elements of `x` to the rows of `y`, and elements of the depth below that to the columns of `y`, and so on until the elements of the deepest valid depth of `x` are cast to the last dimension of `y`.

Similarly, the call
`x <- cast_dim2hier(y, in2out = FALSE)`
will cast the rows of `y` to the surface-level elements of `x`, and the columns of `y` to one depth below that, and so on until the last dimension of `y` is cast to the inner most elements of `x`.

Consider the nested list `x` with a depth of `3`, and the recursive array `y` with 3 dimensions, where their relationship can described with the following code:
`y <- cast_hier2dim(x, in2out = FALSE)`
`x <- cast_dim2hier(y, in2out = FALSE)`.
Then it holds that :

`x[[i]][[j]][[k]]` corresponds to `y[[i, j, k]]`,
$\forall$(`i`, `j`, `k`) , provided `x[[i]][[j]][[k]]` exists.

`acast` *Simple and Fast Casting/Pivoting of an Array*

Description

The `acast()` function spreads subsets of an array margin over a new dimension.
Written in 'C' and 'C++' for high speed and memory efficiency.

Roughly speaking, `acast()` can be thought of as the "array" analogy to `data.table::dcast()`.
But note 2 important differences:

- `acast()` works on arrays instead of data.tables.
- `acast()` casts into a completely new dimension (namely `ndim(x) + 1`), instead of casting into new columns.

Usage

```
acast(x, ...)

## Default S3 method:
acast(
  x,
  margin,
  grp,
  fill = FALSE,
  fill_val = if (is.atomic(x)) NA else list(NULL),
  ...
)
```

Arguments

`x`	an atomic or recursive array.
`...`	further arguments passed to or from methods.
`margin`	a scalar integer, specifying the margin to cast from.
`grp`	a factor, where `length(grp) == dim(x)[margin]`, with at least 2 unique values, specifying which indices of `dim(x)[margin]` belong to which group. Each group will be cast onto a separate index of dimension `ndim(x) + 1`. Unused levels of `grp` will be dropped. Any `NA` values or levels found in `grp` will result in an error.

`fill` Boolean.
When factor `grp` is unbalanced (i.e. has unequally sized groups) the result will be an array where some slices have missing values, which need to be filled. If `fill = TRUE`, an unbalanced `grp` factor is allowed, and missing values will be filled with `fill_val`.
If `fill = FALSE` (default), an unbalanced `grp` factor is not allowed, and providing an unbalanced factor for `grp` produces an error.
When `x` has type of `raw`, unbalanced `grp` is never allowed.

`fill_val` scalar of the same type of `x`, giving value to use to fill in the gaps when `fill = TRUE`.
The `fill_val` argument is ignored when `fill = FALSE` or when `x` has type of `raw`.

Details

For the sake of illustration, consider a matrix `x` and a grouping factor `grp`.
Let the integer scalar `k` represent a group in `grp`, such that $k \in$ `1:nlevels(grp)`.
Then the code
`out <- acast(x, margin = 1, grp = grp)`
essentially performs the following for every group `k`:

- copy-paste the subset `x[grp == k, ]` to the subset `out[, , k]`.

Please see the examples section to get a good idea on how this function casts an array.

Value

An array with the following properties:

- the number of dimensions of the output array is equal to `ndim(x) + 1`;
- the dimensions of the output array is equal to `c(dim(x), max(tabulate(grp))`;
- the `dimnames` of the output array is equal to `c(dimnames(x), list(levels(grp)))`.

Back transformation

From the casted array,
`out <- acast(x, margin, grp)`,
one can get the original `x` back by using
`back <- asplit(out, ndim(out)) |> bind_array(along = margin)`.
Note, however, the following about the back-transformed array `back`:

- `back` will be ordered by `grp` along dimension `margin`;
- if the levels of `grp` did not have equal frequencies, then `dim(back)[margin] > dim(x)[margin]`, and `back` will have more missing values than `x`.

See Also

broadcast_casting

Examples

```
x <- cbind(id = c(rep(1:3, each = 2), 1), grp = c(rep(1:2, 3), 2), val = rnorm(7))
print(x)

grp <- as.factor(x[, 2])
levels(grp) <- c("a", "b")
margin <- 1L

acast(x, margin, grp, fill = TRUE)
```

---

bc.b *Broadcasted Boolean Operations*

Description

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

Please note that these operations will treat the input as `logical`.
Therefore, something like `bc.b(1, 2, "==")` returns `TRUE`, because both `1` and `2` are `TRUE` when treated as `logical`.

For regular relational operators, see bc.rel.

Usage

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

## S4 method for signature 'ANY'
bc.b(x, y, op)
```

Arguments

`x`, `y`	conformable arrays of type `logical`, `numeric`, or `raw`.
`op`	a single string, giving the operator. Supported Boolean operators: &, |, xor, nand, ==, !=, <, >, <=, >=.
`...`	further arguments passed to or from methods.

Details

`bc.b()` efficiently casts the input to logical.
Since the input is treated as logical, the following equalities hold for `bc.b()`:

- "==" is equivalent to `(x & y) | (!x & !y)`, but faster;
- "!=" is equivalent to `xor(x, y)`;
- "<" is equivalent to `(!x & y)`, but faster;
- ">" is equivalent to `(x & !y)`, but faster;
- "<=" is equivalent to `(!x & y) | (y == x)`, but faster;
- ">=" is equivalent to `(x & !y) | (y == x)`, but faster.

Value

A logical array as a result of the broadcasted Boolean operation.

See Also

broadcast_operators

Examples

```
x.dim <- c(4:2)
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(4,1,1))

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.bit *Broadcasted Bit-wise Operations*

Description

The `bc.bit()` function performs broadcasted bit-wise operations on pairs of arrays, where both arrays are of type `raw` or both arrays are of type `integer`.

Usage

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

## S4 method for signature 'ANY'
bc.bit(x, y, op)
```

Arguments

x, y	conformable raw or integer (32 bit) vectors or arrays.
op	a single string, giving the operator. Supported bit-wise operators: &, |, xor, nand, ==, !=, <, >, <=, >=, «, ».
...	further arguments passed to or from methods.

Details

The "&", "|", "xor", and "nand" operators given in `bc.bit()` perform BIT-WISE AND, OR, XOR, and NAND operations, respectively.

The relational operators given in `bc.bit()` perform BIT-WISE relational operations:

- "==" is equivalent to bit-wise `(x & y) | (!x & !y)`, but faster;
- "!=" is equivalent to bit-wise `xor(x, y)`;
- "<" is equivalent to bit-wise `(!x & y)`, but faster;
- ">" is equivalent to bit-wise `(x & !y)`, but faster;
- "<=" is equivalent to bit-wise `(!x & y) | (y == x)`, but faster;
- ">=" is equivalent to bit-wise `(x & !y) | (y == x)`, but faster.

The "«" and "»" operators perform bit-wise left-shift and right-shift, respectively, on `x` by unit `y`.
For these shift operations, `y` being larger than the number of bits of `x` results in an error.
Shift operations are only supported for type of `raw`.

Value

For bit-wise operators:
An array of the same type as `x` and `y`, as a result of the broadcasted bit-wise operation.

See Also

broadcast_operators

Examples

```
x.dim <- c(4:2)
x.len <- prod(x.dim)
x.data <- as.raw(0:10)
y.data <- as.raw(10:0)
x <- array(x.data, x.dim)
y <- array(y.data, c(4,1,1))

bc.bit(x, y, "&")
bc.bit(x, y, "|")
bc.bit(x, y, "xor")
```

---

bc.cplx *Broadcasted Complex Numeric Operations*

Description

The `bc.cplx()` function performs broadcasted complex numeric operations on pairs of 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, ...)

## S4 method for signature 'ANY'
bc.cplx(x, y, op)
```

Arguments

`x, y`	conformable atomic arrays of type `complex`.
`op`	a single string, giving the operator. Supported arithmetic operators: +, -, *, /. Supported relational operators: ==, !=.
`...`	further arguments passed to or from methods.

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.

See Also

broadcast_operators

Examples

```
x.dim <- c(4:2)
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(4,1,1))

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.d *Broadcasted Decimal Numeric Operations*

Description

The `bc.d()` function performs broadcasted decimal numeric operations on 2 numeric or logical arrays.

Usage

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

## S4 method for signature 'ANY'
bc.d(x, y, op, tol = sqrt(.Machine$double.eps))
```

Arguments

x, y	conformable logical or numeric arrays.
op	a single string, giving the operator. Supported arithmetic operators: +, -, *, /, ^, pmin, pmax. Supported relational operators: ==, !=, <, >, <=, >=, d==, d!=, d<, d>, d<=, d>=.
...	further arguments passed to or from methods.
tol	a single number between 0 and 0.1, giving the machine tolerance 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 decimal arithmetic operation.

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

See Also

broadcast_operators

Examples

```
x.dim <- c(4:2)
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(4,1,1))

bc.d(x, y, "+")
bc.d(x, y, "-")
bc.d(x, y, "*")
bc.d(x, y, "/")
```

```
bc.d(x, y, "^")

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

`bc.i` *Broadcasted Integer Numeric Operations with Extra Overflow Protection*

Description

The `bc.i()` function performs broadcasted integer numeric operations on 2 numeric or logical arrays.

Please note that these operations will treat the input as (`double` typed) integers, and will efficiently truncate when necessary.
Therefore, something like `bc.i(1, 1.5, "==")` returns `TRUE`, because `trunc(1.5)` equals `1`.

For regular relational operators, see bc.rel.

Usage

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

## S4 method for signature 'ANY'
bc.i(x, y, op)
```

Arguments

`x`, `y`	conformable logical or numeric arrays.
`op`	a single string, giving the operator. Supported arithmetic operators: +, -, *, gcd, %%, %/%, ^, pmin, pmax. Supported relational operators: ==, !=, <, >, <=, >=. The "gcd" operator performs the Greatest Common Divisor" operation, using the Euclidean algorithm.
`...`	further arguments passed to or from methods.

Value

For arithmetic operators:
A numeric array of whole numbers, as a result of the broadcasted arithmetic operation.
Base 'R' supports integers from `-2^53` to `2^53`, which thus range from approximately `-9` quadrillion to `+9` quadrillion.
Values outside of this range will be returned as `-Inf` or `Inf`, as an extra protection against integer overflow.

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

See Also

broadcast_operators

Examples

```
x.dim <- c(4:2)
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(4,1,1))

bc.i(x, y, "+")
bc.i(x, y, "-")
bc.i(x, y, "*")
bc.i(x, y, "gcd") # greatest common divisor
bc.i(x, y, "^")

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

---

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, ...)

## S4 method for signature 'ANY'
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.
...	further arguments passed to or from methods.

Value

A recursive array.

See Also

broadcast_operators

Examples

```
x.dim <- 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.raw` *Broadcasted Operations that Take Raw Arrays and Return Raw Arrays*

Description

The `bc.raw()` function performs broadcasted operations on arrays of type `raw`, and the return type is **always** `raw`.

For bit-wise operations, use bc.bit.
For relational operations with logical (`TRUE`/`FALSE`/`NA`) results, use bc.rel.

Usage

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

## S4 method for signature 'ANY'
bc.raw(x, y, op)
```

Arguments

`x`, `y`	conformable raw vectors or arrays.
`op`	a single string, giving the operator. Supported operators: ==, !=, <, >, <=, >=, pmin, pmax, diff. The relational operators work the same as in bc.rel, but with the following difference: a `TRUE` result is replaced with `01`, and a `FALSE` result is replaced with `00`. The "diff" operator performs the byte equivalent of `abs(x - y)`.
`...`	further arguments passed to or from methods.

Value

`bc.raw()` **always** returns an array of type `raw`.
For the relational operators, `01` codes for `TRUE` results, and `00` codes for `FALSE` results.

See Also

broadcast_operators

Examples

```
x <- array(
  sample(as.raw(1:100)), c(5, 3, 2)
)
y <- array(
  sample(as.raw(1:100)), c(5, 1, 1)
)
```

```
cond <- bc.raw(x, y, "!=")
print(cond)

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

`bc.rel` *Broadcasted General Relational Operators*

Description

The `bc.rel()` function performs broadcasted general relational operations on 2 arrays.

Usage

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

## S4 method for signature 'ANY'
bc.rel(x, y, op)
```

Arguments

`x, y`	conformable arrays of any atomic type.
`op`	a single string, giving the relational operator. Supported relational operators: ==, !=, <, >, <=, >=.
`...`	further arguments passed to or from methods.

Value

A logical array as a result of the broadcasted general relational operation.

See Also

broadcast_operators

Examples

```
x.dim <- c(4:2)
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(4,1,1))


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

`bc.str` *Broadcasted String Operations*

Description

The `bc.str()` function performs broadcasted string operations on pairs of arrays.

Usage

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

## S4 method for signature 'ANY'
bc.str(x, y, op)
```

Arguments

x, y	conformable atomic arrays of type `character`.
op	a single string, giving the operator. Supported concatenation operators: +. Supported relational operators: ==, !=. Supported distance operators: levenshtein.
...	further arguments passed to or from methods.

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 distance measurement.

References

The 'C++' code for the Levenshtein edit string distance is based on the code found in https://rosettacode.org/wiki/Levenshtein_distance#C++

See Also

broadcast_operators

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 Function to Pair of Arrays with Broadcasting*

Description

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

Usage

```
bcapply(x, y, f, ...)

## S4 method for signature 'ANY'
bcapply(x, y, f, v = NULL)
```

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.
...	further arguments passed to or from methods.
v	either NULL, or single string, giving the scalar type for a single iteration. If NULL (default) or "list", 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; when unsure, leave v at its default value.

Value

An atomic or recursive array with dimensions bc_dim(x, y).
Preserves some of the attributes of x and y similar to broadcasted infix operators, as explained in broadcast_operators.

Examples

```
x.dim <- c(5, 3, 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(1:5, c(5, 1, 1))

f <- function(x, y) strrep(x, y)
bcapply(x, y, f, v = "character")
```

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 or recursive array.

Value

Returns an integer vector giving the broadcasted dimension sizes of the result, or the length of the result if its dimensions will be `NULL`.

Examples

```
x.dim <- c(4:2)
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(4,1,1))

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()` function performs a broadcasted form of `ifelse()`.

Usage

```
bc_ifelse(test, yes, no, ...)

## S4 method for signature 'ANY'
bc_ifelse(test, yes, no)
```

Arguments

test	a vector or array, with the type logical, integer, or raw, and a length equal to prod(bc_dim(yes, no)). If yes / no are of type raw, test is not allowed to contain any NAs.
yes, no	conformable arrays of the same type. All atomic types are supported. Recursive arrays of type list are also supported.
...	further arguments passed to or from methods.

Value

The output, here referred to as out, will be an array of the same type as yes and no.
If test has the same dimensions as bc_dim(yes, no), then out will also have the same dimnames as test.
If test is a broadcaster, then out will also be a broadcaster.

After broadcasting yes against no, given any element index i, the following will hold for the output:

- when test[i] == TRUE, out[i] is yes[i];
- when test[i] == FALSE, out[i] is no[i];
- when test[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(5, 3, 2)
x.len <- prod(x.dim)

x <- array(sample(1:100), x.dim)
y <- array(sample(1:100), c(5, 1, 1))

cond <- bc.i(x, y, ">")

bc_ifelse(cond, yes = x^2, no = -y)
```

bind_array *Dimensional Binding of Arrays with Broadcasting*

Description

`bind_array()` binds (atomic/recursive) arrays and (atomic/recursive) matrices.
Allows for broadcasting.

Usage

```
bind_array(
  input,
  along,
  rev = FALSE,
  ndim2bc = 16L,
  name_along = TRUE,
  comnames_from = 1L
)
```

Arguments

`input`	a list of arrays; both atomic and recursive arrays are supported, and can be mixed. If argument `input` has length `0`, or it contains exclusively objects where one or more dimensions are `0`, an error is returned. If `input` has length `1`, `bind_array()` simply returns `input[[1L]]`. `input` may not contain more than `2^16` objects.
`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. 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 = max(lst.ndim(input))`, will create an additional dimension (`N + 1`) and bind the arrays along that new dimension.
`rev`	Boolean, indicating if `along` should be reversed, counting backwards. If `FALSE` (default), `along` works like normally; if `TRUE`, `along` is reversed. I.e. `along = 0, rev = TRUE` is equivalent to `along = N+1, rev = FALSE`; and `along = N+1, rev = TRUE` is equivalent to `along = 0, rev = FALSE`; with `N = max(lst.ndim(input))`.
`ndim2bc`	a single non-negative integer; specify here the maximum number of dimensions that are allowed to be broadcasted when binding arrays. If `ndim2bc = 0L`, **no** broadcasting will be allowed at all.

name_along	Boolean, indicating if dimension along should be named. Please run the code in the examples section to get a demonstration of the naming behaviour.
comnames_from	either an integer scalar or NULL. 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.

Value

An array.

Examples

```
# Simple example ====
x <- array(1:20, c(5, 4))
y <- array(-1:-15, c(5, 3))
z <- array(21:40, c(5, 4))
input <- list(x, y, z)
# column binding:
bind_array(input, 2L)


# Broadcasting example ====
x <- array(1:20, c(5, 4))
y <- array(-1:-5, c(5, 1))
z <- array(21:40, c(5, 4))
input <- list(x, y, z)
bind_array(input, 2L)


# Mixing types ====
# 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))
) |> matrix(4, 3, byrow = TRUE)
dimnames(x) <- list(letters[1:4], LETTERS[1:3])
```

```
print(x)

y <- matrix(1:12, 4, 3)
print(y)
z <- matrix(letters[1:12], c(4, 3))

# column-binding:
input <- list(x = x, y = y, z = z)
bind_array(input, along = 2L)



# Illustrating `along` argument ====
# using recursive arrays for clearer visual distinction
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 (i.e. rows)
bind_array(input, along = 2L)
bind_array(input, along = 3L) # bind on new dimension after last

bind_array(input, along = 0L, TRUE) # binds on new dimension after last
bind_array(input, along = 1L, TRUE) # binds on last dimension (i.e. columns)
bind_array(input, along = 2L, TRUE)
bind_array(input, along = 3L, TRUE) # bind on new dimension before first



# binding, with 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



# Illustrating `name_along` ====
x <- array(1:20, c(5, 3), list(NULL, LETTERS[1:3]))
y <- array(-1:-20, c(5, 3))
z <- array(-1:-20, c(5, 3))

bind_array(list(a = x, b = y, z), 2L)
bind_array(list(x, y, z), 2L)
bind_array(list(a = unname(x), b = y, c = z), 2L)
bind_array(list(x, a = y, b = z), 2L)
input <- list(x, y, z)
names(input) <- c("", NA, "")
bind_array(input, 2L)
```

broadcaster *Check or Set if an Array is a Broadcaster*

Description

`broadcaster()` checks if an array has the "broadcaster" attribute.
`broadcaster()<-` sets or un-sets the class attribute "broadcaster" on an array.

The `broadcaster` class attribute exists purely to overload the arithmetic, Boolean, bit-wise, and relational infix operators, to support broadcasting.
This makes mathematical expressions with multiple variables, where precedence may be important, far more convenient.
Like in the following calculation:
`x / (y + z)`

See broadcast_operators for more information.

Usage

```
broadcaster(x)

broadcaster(x) <- value
```

Arguments

`x`	object to check or set. Only S3 vectors and arrays are supported, and only up to 16 dimensions.
`value`	set to `TRUE` to make an array a broadcaster, or `FALSE` to remove the broadcaster class attribute from an array.

Value

For `broadcaster()`:
`TRUE` if an array or vector is a broadcaster, or `FALSE` if it is not.

For `broadcaster()<-`:
Returns nothing, but sets (if right hand side is `TRUE`) or removes (if right hand side is `FALSE`) the "broadcaster" class attribute.

Examples

```
# maths ====

x <- 1:10
```

```
broadcaster(x) <- TRUE

y <- 1:10
broadcaster(y) <- TRUE

x + y / x
x + 1 # mathematically equivalent to the above, since x == y
(x + y) / x
2 * x/x # mathematically equivalent to the above, since x == y

dim(x) <- c(10, 1)
dim(y) <- c(1, 10)

x + y / x
(x + y) / x

(x + y) * x


# relational operators ====
x <- 1:10
y <- array(1:10, c(1, 10))
broadcaster(x) <- TRUE
broadcaster(y) <- TRUE

x == y
x != y
x < y
x > y
x <= y
x >= y
```

`cast_dim2flat` *Cast Dimensional List into a Flattened List*

Description

`cast_dim2flat()` casts a dimensional list (i.e. recursive array) into a flat list (i.e. recursive vector), but with names that indicate the original dimensional positions of the elements.

Primary purpose for this function is to facilitate printing or summarizing dimensional lists.

Usage

```
cast_dim2flat(x, ...)

## Default S3 method:
cast_dim2flat(x, ...)
```

Arguments

x	a list
...	further arguments passed to or from methods.

Value

A flattened list, with names that indicate the original dimensional positions of the elements.

See Also

broadcast_casting

Examples

```
# Example 1: Basics ====
x <- list(
  group1 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    ),
    class2 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  ),
  group2 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    ),
    class2 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  )
)

# predict what dimensions `x` would have if casted as dimensional:
hier2dim(x)
```

```
x2 <- cast_hier2dim(x) # cast as dimensional

# since the original list uses the same names for all elements within the same depth,
# dimnames can be set easily:
dimnames(x2) <- list( # go from deep names to surface names
  c("height", "weight", "sex"),
  c("class1", "class2"),
  c("group1", "group2")
)

print(x2) # very compact, maybe too compact...?

# print a small portion of the list, but less compact:
cast_dim2flat(x2[, 1:2, "group1", drop = FALSE])


# Example 2: Cast from outside to inside ====
x <- list(
  group1 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    ),
    class2 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  ),
  group2 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    ),
    class2 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  )
)

# by default, `in2out = TRUE`;
# for this example, `in2out = FALSE` is used

# predict what dimensions `x` would have if casted as dimensional:
hier2dim(x, in2out = FALSE)

x2 <- cast_hier2dim(x, in2out = FALSE) # cast as dimensional

# since the original list uses the same names for all elements within the same depth,
```

```
# dimnames can be set easily:
# because in2out = FALSE, go from the shallow names to the deeper names:
dimnames(x2) <- list( # notice the order here is reversed, because in2out = FALSE
  c("group1", "group2"),
  c("class1", "class2"),
  c("height", "weight", "sex")
)

print(x2) # very compact, maybe too compact...?

# print a small portion of the list, but less compact:
cast_dim2flat(x2["group1", 1:2, , drop = FALSE])



# Example 3: padding ====

# For Example 3, take the same list as before, but remove x$group1$class2:

x <- list(
  group1 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  ),
  group2 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    ),
    class2 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  )
)


hier2dim(x) # as indicated here, dimension 2 (i.e. columns) will have padding

# casting this to a dimensional list will resulting in padding with `NULL`:
x2 <- cast_hier2dim(x)
print(x2)
# The `NULL` values are added for padding.
# This is because all slices of the same dimension need to have the same number of elements.
# For example, all rows need to have the same number of columns.

# one can also use custom padding:
x2 <- cast_hier2dim(x, padding = list(~ "this is padding"))
```

```
print(x2)

dimnames(x2) <- list(
  c("height", "weight", "sex"),
  c("class1", "class2"),
  c("group1", "group2")
)

print(x2)

cast_dim2flat(x2[1:2, , , drop = FALSE])

# we can also use in2out = FALSE:
x2 <- cast_hier2dim(x, in2out = FALSE, padding = list(~ "this is padding"))
dimnames(x2) <- list( # notice the order here is reversed, because in2out = FALSE
  c("group1", "group2"),
  c("class1", "class2"),
  c("height", "weight", "sex")

)
print(x2)

cast_dim2flat(x2[, , 1:2, drop = FALSE])
```

`cast_dim2hier` *Cast Dimensional List into Hierarchical List*

Description

`cast_dim2hier()` casts a dimensional list (i.e. an array of type `list`) into a hierarchical/nested list.

Usage

```
cast_dim2hier(x, ...)

## Default S3 method:
cast_dim2hier(x, in2out = TRUE, distr.names = FALSE, ...)
```

Arguments

`x`	an array of type `list`.
`...`	further arguments passed to or from methods.
`in2out`	see broadcast_casting.

distr.names — TRUE or FALSE, indicating if dimnames from x should be distributed over the nested elements of the output.
See examples section for demonstration.

Value

A nested list.

See Also

broadcast_casting

Examples

```
x <- array(c(as.list(1:24), as.list(letters)), 4:2)
dimnames(x) <- list(
  letters[1:4],
  LETTERS[1:3],
  month.abb[1:2]
)
print(x)


# cast `x` from in to out, and distribute names:
x2 <- cast_dim2hier(x, distr.names = TRUE)
head(x2, n = 2)

# cast `x` from out to in, and distribute names:
x2 <- cast_dim2hier(x, in2out = FALSE, distr.names = TRUE)
head(x2, n = 2)
```

---

cast_hier2dim *Cast Hierarchical List into Dimensional list*

Description

cast_hier2dim() casts a hierarchical/nested list into a dimensional list (i.e. an array of type list). hier2dim() takes a hierarchical/nested list, and predicts what dimensions the list would have, if casted by the cast_hier2dim() function.

Usage

```
cast_hier2dim(x, ...)

hier2dim(x, ...)

## Default S3 method:
cast_hier2dim(
  x,
  in2out = TRUE,
  maxdepth = 16L,
  recurse_classed = FALSE,
  padding = list(NULL),
  ...
)

## Default S3 method:
hier2dim(x, in2out = TRUE, maxdepth = 16L, recurse_classed = FALSE, ...)
```

Arguments

x	a nested list. If x has redundant nesting, it is advisable (though not necessary) to reduce the redundant nesting using dropnests.
...	further arguments passed to or from methods.
in2out	see broadcast_casting.
maxdepth	a single, positive integer, giving the maximum depth to recurse into the list. The surface-level elements of a list is depth 1.
recurse_classed	TRUE or FALSE, indicating if the function should also recurse through classed lists within x, like data.frames.
padding	a list of length 1, giving the padding value to use when padding is required. Padding is used to ensure every all slices of the same dimension in the output have equal number of elements (for example, all rows must have the same number of columns).

Value

For hier2dim():
An integer vector, giving the dimensions x would have, if casted by cast_hier2dim().
The names of the output indicates if padding is required (name "padding"), or no padding is required (no name) for that dimension;
Padding will be required if not all list-elements at a certain depth have the same length.

For cast_hier2dim():

An array of type `list`, with the dimensions given by `hier2dim()`.
If the output needs padding (indicated by `hier2dim()`), the output will have more elements than `x`, filled with a padding value (as specified in the `padding` argument).

See Also

broadcast_casting

Examples

```
# Example 1: Basics ====
x <- list(
  group1 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    ),
    class2 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  ),
  group2 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    ),
    class2 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  )
)

# predict what dimensions `x` would have if casted as dimensional:
hier2dim(x)

x2 <- cast_hier2dim(x) # cast as dimensional

# since the original list uses the same names for all elements within the same depth,
# dimnames can be set easily:
dimnames(x2) <- list( # go from deep names to surface names
  c("height", "weight", "sex"),
  c("class1", "class2"),
```

```
  c("group1", "group2")
)

print(x2) # very compact, maybe too compact...?

# print a small portion of the list, but less compact:
cast_dim2flat(x2[, 1:2, "group1", drop = FALSE])


# Example 2: Cast from outside to inside ====
x <- list(
  group1 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    ),
    class2 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  ),
  group2 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    ),
    class2 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  )
)

# by default, `in2out = TRUE`;
# for this example, `in2out = FALSE` is used

# predict what dimensions `x` would have if casted as dimensional:
hier2dim(x, in2out = FALSE)

x2 <- cast_hier2dim(x, in2out = FALSE) # cast as dimensional

# since the original list uses the same names for all elements within the same depth,
# dimnames can be set easily:
# because in2out = FALSE, go from the shallow names to the deeper names:
dimnames(x2) <- list( # notice the order here is reversed, because in2out = FALSE
  c("group1", "group2"),
  c("class1", "class2"),
  c("height", "weight", "sex")
)
```

```
print(x2) # very compact, maybe too compact...?

# print a small portion of the list, but less compact:
cast_dim2flat(x2["group1", 1:2, , drop = FALSE])



# Example 3: padding ====

# For Example 3, take the same list as before, but remove x$group1$class2:

x <- list(
  group1 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  ),
  group2 = list(
    class1 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    ),
    class2 = list(
      height = rnorm(10, 170),
      weight = rnorm(10, 80),
      sex = sample(c("M", "F", NA), 10, TRUE)
    )
  )
)


hier2dim(x) # as indicated here, dimension 2 (i.e. columns) will have padding

# casting this to a dimensional list will resulting in padding with `NULL`:
x2 <- cast_hier2dim(x)
print(x2)
# The `NULL` values are added for padding.
# This is because all slices of the same dimension need to have the same number of elements.
# For example, all rows need to have the same number of columns.

# one can also use custom padding:
x2 <- cast_hier2dim(x, padding = list(~ "this is padding"))
print(x2)

dimnames(x2) <- list(
  c("height", "weight", "sex"),
  c("class1", "class2"),
  c("group1", "group2")
)
```

```
print(x2)

cast_dim2flat(x2[1:2, , , drop = FALSE])

# we can also use in2out = FALSE:
x2 <- cast_hier2dim(x, in2out = FALSE, padding = list(~ "this is padding"))
dimnames(x2) <- list( # notice the order here is reversed, because in2out = FALSE
  c("group1", "group2"),
  c("class1", "class2"),
  c("height", "weight", "sex")

)
print(x2)

cast_dim2flat(x2[, , 1:2, drop = FALSE])
```

`dropnests` *Drop Redundant List Nesting*

Description

`dropnests()` drops redundant nesting of a list.
It is the hierarchical equivalent to the dimensional `base::drop()` function.

Usage

```
dropnests(x, ...)

## Default S3 method:
dropnests(x, maxdepth = 16L, recurse_classed = FALSE, ...)
```

Arguments

`x`	a list
`...`	further arguments passed to or from methods.
`maxdepth`	a single, positive integer, giving the maximum depth to recurse into the list. The surface-level elements of a list is depth 1. `dropnests(x, maxdepth = 1)` will return `x` unchanged.
`recurse_classed`	`TRUE` or `FALSE`, indicating if the function should also recurse through classed lists within `x`, like data.frames.

Value

A list without redundant nesting.
Attributes are preserved.

See Also

broadcast_casting

Examples

```
x <- list(
  a = list(list(list(list(1:10)))),
  b = list(1:10)
)
print(x)


dropnests(x)


# recurse_classed demonstration ====
x <- list(
  a = list(list(list(list(1:10)))),
  b = data.frame(month.abb, month.name),
  c = data.frame(month.abb)
)


dropnests(x) # by default, recurse_classed = FALSE

dropnests(x, recurse_classed = TRUE)


# maxdepth demonstration ====
x <- list(
  a = list(list(list(list(1:10)))),
  b = list(1:10)
)
print(x)


dropnests(x) # by default, maxdepth = 16

dropnests(x, maxdepth = 3L)

dropnests(x, maxdepth = 1L) # returns `x` unchanged
```

`linear_algebra_stats` *Simple Linear Algebra Functions for Statistics*

Description

'broadcast' provides some simple Linear Algebra Functions for Statistics:
`cinv()`;
`sd_lc()`.

Usage

```
cinv(x)

sd_lc(X, vc, bad_rp = NaN)
```

Arguments

`x`	a real symmetric positive-definite square matrix.
`X`	a numeric (or logical) matrix of multipliers/constants
`vc`	the variance-covariance matrix for the (correlated) random variables.
`bad_rp`	if `vc` is not a Positive (semi-) Definite matrix, give here the value to replace bad standard deviations with.

Details

cinv()
`cinv()` computes the Choleski inverse of a real symmetric positive-definite square matrix.

sd_lc()
Given the linear combination `X %*% b`, where:

- `X` is a matrix of multipliers/constants;
- `b` is a vector of (correlated) random variables;
- `vc` is the symmetric variance-covariance matrix for `b`;

`sd_lc(X, vc)` computes the standard deviations for the linear combination `X %*% b`, without making needless copies.
`sd_lc(X, vc)` will use **much** less memory than a base 'R' approach.
`sd_lc(X, vc)` may possibly, but not necessarily, be faster than a base 'R' approach (depending on the Linear Algebra Library used for base 'R').

Value

For `cinv()`:
A matrix.

For `sd_lc()`:
A vector of standard deviations.

References

John A. Rice (2007), *Mathematical Statistics and Data Analysis* (6th Edition)

See Also

chol, chol2inv

Examples

```
vc <- datasets::ability.cov$cov
X <- matrix(rnorm(100), 100, ncol(vc))

solve(vc)
cinv(vc) # faster than `solve()`, but only works on positive definite matrices
all(round(solve(vc), 6) == round(cinv(vc), 6)) # they're the same

sd_lc(X, vc)
```

---

ndim *Get the Number of Dimensions of an Array*

Description

`ndim()` returns the number of dimensions of an object.
`lst.ndim()` returns the number of dimensions of every list-element.

Usage

```
ndim(x)

lst.ndim(x)
```

Arguments

x	a vector or array (for ndim()), or a list of vectors/arrays (for lst.ndim()).

Value

For ndim(): an integer scalar.
For lst.ndim(): an integer vector, with the same length, names and dimensions as x.

Examples

```
# matrix example ====
x <- list(
  array(1:10, 10),
  array(1:10, c(2, 5)),
  array(c(letters, NA), c(3,3,3))
)
lst.ndim(x)
```

---

rep_dim *Replicate Array Dimensions*

Description

The rep_dim() function replicates array dimensions until the specified dimension sizes are reached, and returns the array.

The various broadcasting functions recycle array dimensions virtually, meaning little to no additional memory is needed.
The rep_dim() function, however, physically replicates the dimensions of an array (and thus actually occupies additional memory space).

Usage

```
rep_dim(x, tdim)
```

Arguments

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

Value

Returns the replicated 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)

rep_dim(x, c(3,3,2)) # replicate to larger size
```

`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_cplx()`: converts object to atomic type `complex`.
- `as_chr()`: converts object to atomic type `character`.
- `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