Package: ParmOff (via r-universe)

July 4, 2026

Type Package

Title Complex Argument Matching

Version 1.0.3

Date 2026-05-27

Description Provides a flexible framework for matching and validating function arguments against user-defined expectations, including named and positional handling, range limits, and logarithmic transforms. It is designed to simplify robust argument processing in reusable code while keeping function interfaces clear, predictable, and easy to extend.

License LGPL-3

Encoding UTF-8

LazyData true

Imports checkmate

Suggests testthat (>= 3.0.0), knitr, rmarkdown, magicaxis

VignetteBuilder knitr

Config/testthat/edition 3

Repository https://asgr.r-universe.dev

Date/Publication 2026-05-27 09:06:50 UTC

RemoteUrl https://github.com/asgr/parmoff

RemoteRef HEAD

RemoteSha 31273706baf080cc46480a4f660eef9228f9cd03

Contents

ParmLim *Flexible lower/upper limiting for vectors and lists*

Description

Helper functions to apply lower and upper limits to vectors or lists (including nested lists). These are used internally by ParmOff, but can also be used directly for generic list-processing workflows.

Usage

```
ParmLimLo(x, lower, verbose = FALSE)
ParmLimHi(x, upper, verbose = FALSE)
ParmLimBoth(x, lower, upper, verbose = FALSE)
```

Arguments

x	Object to be limited. Usually a numeric vector/scalar, or a list containing numeric leaves. Nested lists are supported.
lower	Lower limit specification. Can be scalar/vector-like, or list-like to mirror all or part of 'x'. Named list elements are matched by name where available. If 'x' is a list and 'lower' is vector then 'lower' is applied to all list elements. Almost always if 'x' is named list then 'lower' should also be a named list (potentially just a subset).
upper	Upper limit specification. Can be scalar/vector-like, or list-like to mirror all or part of 'x'. Named list elements are matched by name where available. If 'x' is a list and 'upper' is vector then 'upper' is applied to all list elements. Almost always if 'x' is named list then 'upper' should also be a named list (potentially just a subset).
verbose	Logical; if TRUE, prints a message for each element where a limit actually changes the value. The message shows the parameter name and its before/after values. Output is only printed for elements that are simple scalars, vectors of length ≤ 20, or matrices of size $\leq 10 \times 10$. Default is FALSE.

Details

ParmLimLo applies pmax recursively; ParmLimHi applies pmin recursively. ParmLimBoth first applies lower limits (via ParmLimLo), then upper limits (via ParmLimHi).

Behaviour when x is a list

The bound is applied recursively to each child element. The child bound is determined as follows, in order:

1. **Bound is a named list**: each child of x is matched by name. Children whose names appear in the bound list receive the corresponding sub-bound; children with no matching name in the bound are left unchanged. This means you only need to specify bounds for the children you care about.

2. **Bound is an unnamed list with one element**: that single element is broadcast to every child of `x`.
3. **Bound is an unnamed list with multiple elements**: children are matched by position (index). Children beyond the length of the bound list are left unchanged.
4. **Bound is a scalar or atomic vector (not a list)**: the bound value is broadcast to every leaf in the entire tree, no matter how deeply nested.

If the bound is a list but `x` is an *atomic* vector (a leaf node), the bound is silently ignored and `x` is returned unchanged, because a list bound has no meaning at a scalar leaf.

Behaviour when `x` is an atomic vector

1. **Both `x` and bound are named**: elements are aligned by name. Elements of `x` whose names have no corresponding entry in the bound (after name alignment) are left unchanged.
2. **Bound is unnamed (scalar or vector)**: applied directly via `pmax` / `pmin`, which recycles bound across the elements of `x` in the standard R manner.
3. **Bound is a list**: ignored; `x` is returned unchanged.

Verbose output

When `verbose = TRUE`, a `message` is emitted for each named top-level element of `x` that is actually changed by the limiting operation. The message shows the element name and its before and after values. To keep output readable, verbose output is suppressed for elements that are atomic vectors longer than 20 or matrices larger than 10×10. Unnamed or deeply nested structures are not reported at sub-list level.

Value

An object with the same overall structure as `x`, with limits applied where possible.

Author(s)

Aaron Robotham

See Also

`ParmOff`, `pmax`, `pmin`

Examples

```
# ------------------------------------------------------------------------
# 1.  Pure vectors
# ------------------------------------------------------------------------

# Unnamed vector: scalar bound applied element-wise
ParmLimLo(c(-3, 0, 5), lower = 0)      # 0  0  5
ParmLimHi(c(-3, 0, 5), upper = 2)      # -3 0  2
ParmLimBoth(c(-3, 0, 5), lower = 0, upper = 2)  # 0  0  2

# Named vector: named bound aligned by name; unmatched elements unchanged
x <- c(a = -5, b = 10, c = 3)
```

```
ParmLimLo(x, lower = c(a = 0))          # a=0, b and c unchanged
ParmLimHi(x, upper = c(b = 7))          # b=7, a and c unchanged
ParmLimBoth(x, lower = c(a = 0), upper = c(b = 7))  # a=0, b=7, c unchanged

# Named vector: unnamed bound recycled element-wise (pmax/pmin semantics)
ParmLimLo(c(a = -1, b = 2, c = -3), lower = c(0, 0, 0))  # 0 2 0

# ------------------------------------------------------------------------
# 2.  Simple named list
# ------------------------------------------------------------------------

x <- list(a = -1, b = 5)

# Named list bound: only named children are affected
ParmLimLo(x, lower = list(a = 0, b = 3))   # a=0,  b=5 (already >= 3)
ParmLimHi(x, upper = list(a = 1, b = 4))   # a=-1, b=4
ParmLimBoth(x,
  lower = list(a = 0, b = 3),
  upper = list(a = 1, b = 4)
)

# Partial bound: only 'a' has a lower bound; 'b' is untouched
ParmLimLo(x, lower = list(a = 0))   # a=0, b=5

# Scalar broadcast: applies the same bound to every list element
ParmLimLo(x, lower = 0)    # a=0,  b=5

# ------------------------------------------------------------------------
# 3.  Verbose output for debugging
# ------------------------------------------------------------------------

# Print before/after values when clamping actually changes something
ParmLimLo(list(a = -1, b = 5), lower = list(a = 0), verbose = TRUE)
# Message emitted (multi-line):
#   Lower limit imposed on 'a'
#     before: -1
#     after:  0
# 'b' is not reported because it is already above its lower bound

ParmLimBoth(list(x = -2, y = 15), lower = list(x = 0), upper = list(y = 10),
            verbose = TRUE)

# ------------------------------------------------------------------------
# 4.  Nested list (2 levels)
# ------------------------------------------------------------------------

x <- list(a = -1, b = list(c = 5, d = -3), e = 9)

# Named nested bound — only the named children at each level are bounded
ParmLimLo(x, lower = list(a = 0, b = list(d = -2)))
# a=0, b$c unchanged (no bound), b$d=-2, e unchanged

ParmLimHi(x, upper = list(b = list(c = 4), e = 8))
```

```
# a unchanged, b$c=4, b$d unchanged, e=8

# Scalar broadcast to all leaves, regardless of nesting depth
ParmLimLo(x, lower = 0)   # a=0, b$c=5 (already >=0), b$d=0, e=9

# ---------------------------------------------------------------------------
# 5.  Deeply nested list (3 levels)
# ---------------------------------------------------------------------------

deep <- list(p = list(q = list(r = -5, s = 20), t = 3), u = -1)

lower <- list(p = list(q = list(r = 0), t = 0), u = 0)
upper <- list(p = list(q = list(s = 10), t = 5))

ParmLimBoth(deep, lower = lower, upper = upper)
# p$q$r: clamped to 0 (was -5)
# p$q$s: clamped to 10 (was 20)
# p$t:   3  (above lower 0, below upper 5 — unchanged)
# u:     0  (clamped to lower 0)

# Scalar bounds broadcast all the way to the deepest leaves
ParmLimBoth(deep, lower = 0, upper = 10)

# ---------------------------------------------------------------------------
# 6.  ParmOff integration
# ---------------------------------------------------------------------------

model <- function(x, y, z) x + y + z

# Clamp individual arguments before calling the model
ParmOff(model, list(x = -1, y = 20, z = 3),
        .lower = list(x = 0, y = 0),
        .upper = list(y = 10))
# x: 0, y: 10, z: 3  -> sum = 13

# With verbose output showing what was clamped
ParmOff(model, list(x = -1, y = 20, z = 3),
        .lower = list(x = 0, y = 0),
        .upper = list(y = 10),
        .verbose = TRUE)
```

ParmLog *Log or unlog selected elements of a list*

Description

Helper functions to apply a log or inverse-log transformation to selected elements of a list. Elements are selected either by a logical vector or by a character vector of names (the same interface used throughout the ParmOff family). The structure of each element (matrix, array, vector, etc.) is preserved in the output.

Usage

```
ParmLog(x, logged, log_type = 'log10', verbose = FALSE)
ParmUnLog(x, logged, log_type = 'log10', verbose = FALSE)
```

Arguments

x	List whose elements are to be (un)logged.
logged	Logical vector (same length as x) or character vector of element names. For a logical vector, elements corresponding to TRUE are transformed. For a character vector, elements whose names appear in logged are transformed.
log_type	Character scalar. Use 'log10' (default) for base-10 transformations (log10 / 10^x), 'ln' for natural-log transformations (log / exp) or 'log2' for base-2 transformations (log2 / 2^x).
verbose	Logical; if TRUE, prints a message for each element that is transformed, showing its name and before/after values. Output is only printed for elements that are simple scalars, vectors of length ≤ 20, or matrices of size $\leq 10 \times 10$. Default is FALSE.

Details

ParmLog applies the forward log transformation (log10 or log) to the selected elements. ParmUnLog applies the inverse (10^x, exp, 2^x).

Internally, lapply is used so that the list structure is never simplified. R's vectorised arithmetic (log10, exp, log2) preserves the dimensions of matrices and arrays, so those structures are returned unchanged in shape.

ParmUnLog is used internally by ParmOff to back-transform arguments listed in .logged. When ParmOff is called with .verbose = TRUE, that flag is forwarded to ParmUnLog so that de-logging transformations are reported automatically.

Value

A copy of x with the selected elements transformed.

Author(s)

Aaron Robotham

See Also

ParmOff, ParmLimLo, log10, log, exp

Examples

```
# --- character-vector selection ---
params <- list(a = 100, b = 10, c = 5)

# Log two elements (base 10)
ParmLog(params, logged = c("a", "b"))
```

```
# a = log10(100) = 2, b = log10(10) = 1, c unchanged

# Unlog them back
ParmUnLog(ParmLog(params, logged = c("a", "b")), logged = c("a", "b"))

# --- logical-vector selection ---
flags <- c(TRUE, TRUE, FALSE)
ParmLog(params, logged = flags)
ParmUnLog(params, logged = flags, log_type = 'ln')  # natural-log inverse

# --- verbose output for debugging ---
# Print the name and before/after values of each transformed element
ParmLog(params, logged = c("a", "b"), verbose = TRUE)
ParmUnLog(list(a = 2, b = 1, c = 5), logged = c("a", "b"), verbose = TRUE)

# --- matrix elements keep their shape ---
mat_params <- list(mu = 5, cov = matrix(c(100, 0, 0, 100), 2, 2))
ParmLog(mat_params, logged = "cov")   # cov becomes log10 of each entry, still 2x2
```

ParmOff *Powerful Parameter Passing*

Description

Simple interface to allow the user to pass complex lists of arguments into functions, with matching and mis-matching rules.

Usage

```
ParmOff(.func, .args = NULL, .use_args = NULL, .rem_args = NULL, .lower = NULL,
  .upper = NULL, .logged = NULL, .constrain = NULL, .strip = NULL, .quote = TRUE,
  .envir = parent.frame(), .pass_dots = TRUE, .return = 'func', .check = TRUE,
  .bound_raw = TRUE, .log_type = 'log10', .clash = 'first', .verbose = FALSE, ...)
```

Arguments

.func	Function; the function to be executed with provided arguments.
.args	List (or named vector); arguments to potential be passed into '.func'. If not a list, then it will be coerced to one with one element per entry (user needs to make sure this is appropriate and fully named, but usually useful if passing in something like a vector of scalar parameters from a fit etc).
.use_args	Character vector; arguments in '.args' and ... will be restricted to this set. This intervention happens first. The name/s provided should be as per post-stripped (if '.strip' is provided).
.rem_args	Character vector; arguments in '.args' and ... will have these arguments removed. The name/s provided should be as per post-stripped (if '.strip' is provided).

`.lower` Numeric list/vector; a list can be all or part of '`.args`', a vector will be coerced to a list. Named elements are matched by name where available and the specified lower limit applied. This is deliberately more restrictive in use than the lower level `ParmLim` functions.

`.upper` Numeric list/vector; a list can be all or part of '`.args`', a vector will be coerced to a list. Named elements are matched by name where available and the specified upper limit applied. This is deliberately more restrictive in use than the lower level `ParmLim` functions.

`.logged` Character/logical vector; logged arguments in '`.args`' and .... These will be unlogged (using base 10), so only specify if this action is desired. The name/s provided should be as per post-stripped (if '`.strip`' is provided). If providing a logical vector it must also be exactly the same length as '`.args`'. We deliberately do not allow just a positional (which-like) integer vector because it is very easy to create subtle bugs.

`.constrain` Function; an optional final intervention on the arguments run immediately before the '`.func`' evaluation. The input to this function is a list of all the arugments that will be pass into '`.func`', and the output must be all the arguments you want to pass but with any user specific constraints applied. This is really for advanced users. An example of when you might use it would be if you know parameter '`y`' always has to be double '`x`' (see Examples), and if limits have been applied this relationship might break down. Given '`.constrain`' is executed last, you also need to be careful to obey any limits required.

`.strip` Character vector; a string element to strip out of all '`.args`' names. This intervention happens first.

`.quote` Logical; to be passed to '`quote`' argument of `do.call`. Leaving as TRUE is usually a good idea because debugging of large inputs is much easier.

`.envir` Environment; to be passed to '`envir`' argument of `do.call`.

`.pass_dots` Logical; if TRUE (default) and '`.func`' has a ... argument input then unmatched arguments will be passed on (on the implicit assumption these will be matched by a function within '`.func`'). If FALSE or '`.func`' has no ... argument input then only remaining matching arguments will be passed into '`.func`'. This intervention happens last.

`.return` Character scalar; if 'func' (or 'function') then the output of `do.call` on the target '`.func`' with the remaining 'current_args', returning the output only. If 'args' (or 'arg') then a list with the 'current_args' (that meet all the matching critera) and 'ignore_args' (provided but not to be used) are returned. If 'func_args' (or 'func_arg') then both are returned in a list with items 'func_out' and 'args' respectively.

`.check` Logical; should input argument checking be carried out? Obviously safer to say TRUE (default), but in speed critical cases you might want to set to FALSE.

`.bound_raw` Logical; should '`.lower`' and '`.upper`' bounds be applied pre de-logging (via '`.logged`') i.e. TRUE (default), or after de-logging i.e. FALSE.

`.log_type` Character scalar. Use `'log10'` (default) for base-10 transformations (`log10` / `10^x`), `'ln'` for natural-log transformations (`log` / `exp`) or `'log2'` for base-2 transformations (`log2` / `2^x`).

.clash	Character scalar; what to do if multiple '.args' have the same name. Since this is generally not allowed when passing arguments into a function, you generally either want to take the 'first' (default) or 'last' option. There is also the option to do 'nothing', which is mainly for de-bugging purposes. Note ... are only used if their names do not clash with the initial provided '.args', i.e. the latter always takes priority if there is a clash (no matter the setting of '.clash'.
.verbose	Logical; if TRUE it will print to screen the names of the used and ignored arguments, and also forwards the flag to ParmLimLo, ParmLimHi, and ParmUnLog so that lower/upper limit clamping and de-logging operations report the parameter name and its before/after values. Output is only printed for elements that are simple scalars, vectors of length ≤ 20, or matrices of size $\leq 10 \times 10$. If FALSE then no verbose output is printed (unless by the called function itself).
...	Other arguments to be merged with '.args' and processed as discussed above. Note ...are only used if their names do not clash with the initial provided '.args', i.e. the latter always takes priority if there is a clash (no matter the setting of '.clash'.

Details

These events happen in this exact order:

1. '.args' has string elements of names removed if '.strip' is provided.
2. If '.args' and ... are both present then any arguments in ... that match by name are removed ('.args' takes precedence).
3. Merge '.args' and ... into one list (current_args).
4. If '.bound_raw' is TRUE (default): if '.lower' is provided, clamp named current_args to the specified minimum values; if '.upper' is provided, clamp to the specified maximum values.
5. current_args has named elements specified by '.logged' unlogged (using base 'log_type', base 10 by default).
6. If '.bound_raw' is FALSE: if '.lower' is provided, clamp named current_args to the specified minimum values; if '.upper' is provided, clamp to the specified maximum values.
7. If '.use_args' is present then restrict current_args to these arguments.
8. If '.rem_args' is present then remove these arguments from current_args.
9. If ...are not present in the arguments of '.func' or '.pass_dots'=FALSE then restrict the current_args list to only those arguments that appear in the formals of '.func'.
10. If there are any name clashes in the remaining current_args then disambiguate using the setting of '.clash'.
11. If '.constrain' is provided apply additional constraints as specified by that function.
12. Return the output of do.call or the current_args (depending on the setting of '.return'.

Whilst '.args' can be a vector input (see Examples), users need to be careful it is fully named and all of a common type (e.g. numeric in most cases). If in doubt, use a named list since this gives full control over the typing of each list element.

It should be noted that for trivially fast functions, ParmOff add a lot of overhead (see Examples). It is generally designed for convenience working with complex and CPU intensive functions (ones that take seconds rather than microseconds to run).

Value

Return of '.func' with whatever the remaining current_args are after the various layers of passing.

Author(s)

Aaron Robotham

See Also

do.call, formals

Examples

```
#Pass a mixture of an argument list and dots, ignoring conflicting arguments from latter:

example_args = list(col='red', xlab='Test x', ylab='Test y')
ParmOff(plot, example_args, x=sin, xlab='Ignore This')

#Ignore the col argument (if present, which it is):
ParmOff(plot, example_args, .rem_args='col', x=sin, xlab='Ignore This')

#Get the current_args and the ignore_args lists:
ParmOff(plot, example_args, .rem_args='col', x=sin, xlab='Ignore This', .return='args')

#An example of a complex model (note for non complex .args you can use a named vector):
model_ex = function(x,y,z){x * y + z}
input = c(x=1, y=2, z=3, t=4) #the input to pass into .args (note 't' will be ignored)
ParmOff(model_ex, input)

#And now tagging argument 'y' to be unlogged:
ParmOff(model_ex, input, .logged='y')

#Bound argument values before passing:
ParmOff(model_ex, input, .lower=list(x=0, y=0.5), .upper=list(y=1.5, z=2))

#Bounding happens before de-logging, so bounds are in log10 space:
ParmOff(model_ex, input, .logged='y', .lower=list(y=0), .upper=list(y=1))

#Make y constrained to be 2*x:
constrain_func = function(x, y, z){list(x=x, y = 2 * x, z=z)}
ParmOff(model_ex, input, .logged='y', .lower=list(y=0), .upper=list(y=1),
  .constrain=constrain_func)

#Truth in advertising, because of the extra checking ParmOff slows down fast functions:

system.time(for(i in 1:1e4){model_ex(input[1], input[2], input[3])})
arg_list = as.list(input)[1:3]
system.time(for(i in 1:1e4){do.call(model_ex, arg_list)})
system.time(for(i in 1:1e4){ParmOff(model_ex, arg_list)})
system.time(for(i in 1:1e4){ParmOff(model_ex, arg_list, .check=FALSE)})
```

Index