Package 'optimizeR'

July 22, 2025

Title Unified Framework for Numerical Optimizers

Version 1.2.1

Description Provides a unified object-oriented framework for numerical optimizers in R. Allows for both minimization and maximization with any optimizer, optimization over more than one function argument, measuring of computation time, setting a time limit for long optimization tasks.

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.2

Depends R (>= 4.0.0)

Imports checkmate, cli, lbfgsb3c, numDeriv, oeli (>= 0.7.2), pracma, R6, stats, TestFunctions, ucminf

Suggests datasets, ggplot2, rmarkdown, testthat

Config/testthat/edition 3

URL https://loelschlaeger.de/optimizeR/, https://github.com/loelschlaeger/optimizeR/

BugReports https://github.com/loelschlaeger/optimizeR/issues

NeedsCompilation no

Author Lennart Oelschläger [aut, cre] (ORCID: <https://orcid.org/0000-0001-5421-9313>), Marius Ötting [ctb] (ORCID: <https://orcid.org/0000-0002-9373-0365>)

Maintainer Lennart Oelschläger <oelschlaeger.lennart@gmail.com>

Repository CRAN

Date/Publication 2025-06-26 13:40:10 UTC

Contents

`Objective` *Specify objective function object*

Description

The `Objective` object specifies the framework for an objective function for numerical optimization.

Value

An `Objective` object.

Active bindings

`objective_name [character(1)]`
: The label for the objective function.

`fixed_arguments [character(), read-only]`
: The name(s) of the fixed argument(s) (if any).

`seconds [numeric(1)]`
: A time limit in seconds. Computations are interrupted prematurely if `seconds` is exceeded.

No time limit if `seconds = Inf` (the default).

Note the limitations documented in `setTimeLimit`.

`hide_warnings [logical(1)]`
: Hide warnings when evaluating the objective function?

`verbose [logical(1)]`
: Print status messages?

`npar [integer(), read-only]`
: The length of each target argument.

`target [character(), read-only]`
: The argument name(s) that get optimized.

`gradient_specified [logical(1), read-only]`
: Whether a gradient function has been specified via `$set_gradient()`.

`hessian_specified [logical(1), read-only]`
: Whether a Hessian function has been specified via `$set_hessian()`.

Methods

Public methods:

- `Objective$new()`
- `Objective$set_argument()`
- `Objective$get_argument()`
- `Objective$remove_argument()`
- `Objective$set_gradient()`

- Objective$set_hessian()
- Objective$evaluate()
- Objective$evaluate_gradient()
- Objective$evaluate_gradient_numeric()
- Objective$evaluate_hessian()
- Objective$evaluate_hessian_numeric()
- Objective$print()
- Objective$clone()

Method `new()`**:** Creates a new `Objective` object.

Usage:

```
Objective$new(f, target = NULL, npar, ...)
```

Arguments:

`f` [`function`]
: A `function` to be optimized.
It is expected that `f` has at least one `numeric` argument.
Further, it is expected that the return value of `f` is of the structure `numeric(1)`, i.e. a single `numeric` value.

`target` [`character()`]
: The argument name(s) that get optimized.
All target arguments must be `numeric`.
Can be `NULL` (default), then the first function argument is selected.

`npar` [`integer()`]
: The length of each target argument, i.e., the length(s) of the `numeric vector` argument(s) specified by `target`.

`...` Optionally additional function arguments that are fixed during the optimization.

Method `set_argument()`**:** Set a function argument that remains fixed during optimization.

Usage:

```
Objective$set_argument(..., .overwrite = TRUE, .verbose = self$verbose)
```

Arguments:

`...` Optionally additional function arguments that are fixed during the optimization.

`.overwrite` [`logical(1)`]
: Overwrite existing values?

`.verbose` [`logical(1)`]
: Print status messages?

Method `get_argument()`**:** Get a fixed function argument.

Usage:

```
Objective$get_argument(argument_name, .verbose = self$verbose)
```

Arguments:

`argument_name` [`character(1)`]
: A function argument name.

`.verbose [logical(1)]`
Print status messages?

Method `remove_argument()`**:** Remove a fixed function argument.

Usage:

```
Objective$remove_argument(argument_name, .verbose = self$verbose)
```

Arguments:

`argument_name [character(1)]`
A function argument name.

`.verbose [logical(1)]`
Print status messages?

Method `set_gradient()`**:** Set a gradient function.

Usage:

```
Objective$set_gradient(
  gradient,
  target = self$target,
  npar = self$npar,
  ...,
  .verbose = self$verbose
)
```

Arguments:

`gradient [function]`
A `function` that computes the gradient of the objective function `f`.
It is expected that `gradient` has the same call as `f`, and that `gradient` returns a `numeric vector` of length `self$npar`.

`target [character()]`
The argument name(s) that get optimized.
All target arguments must be `numeric`.
Can be `NULL` (default), then the first function argument is selected.

`npar [integer()]`
The length of each target argument, i.e., the length(s) of the `numeric vector` argument(s) specified by `target`.

`...` Optionally additional function arguments that are fixed during the optimization.

`.verbose [logical(1)]`
Print status messages?

Method `set_hessian()`**:** Set a Hessian function.

Usage:

```
Objective$set_hessian(
  hessian,
  target = self$target,
  npar = self$npar,
  ...,
  .verbose = self$verbose
)
```

Arguments:

`hessian [function]`
: A `function` that computes the Hessian of the objective function `f`.
It is expected that `hessian` has the same call as `f`, and that `hessian` returns a `numeric matrix` of dimension `self$npar` times `self$npar`.

`target [character()]`
: The argument name(s) that get optimized.
All target arguments must be `numeric`.
Can be `NULL` (default), then the first function argument is selected.

`npar [integer()]`
: The length of each target argument, i.e., the length(s) of the `numeric vector` argument(s) specified by `target`.

`...` Optionally additional function arguments that are fixed during the optimization.

`.verbose [logical(1)]`
: Print status messages?

Method `evaluate()`**:** Evaluate the objective function.

Usage:

```
Objective$evaluate(
  .at,
  .negate = FALSE,
  .gradient_as_attribute = FALSE,
  .gradient_attribute_name = "gradient",
  .gradient_numeric = FALSE,
  .hessian_as_attribute = FALSE,
  .hessian_attribute_name = "hessian",
  .hessian_numeric = FALSE,
  ...
)
```

Arguments:

`.at [numeric()]`
: The values for the target argument(s), written in a single vector.
Must be of length `sum(self$npar)`.

`.negate [logical(1)]`
: Negate the function return value?

`.gradient_as_attribute [logical(1)]`
: Add the value of the gradient function as an attribute to the output?
The attribute name is defined via the `.gradient_attribute_name` argument.
Ignored if `$gradient_specified` and `.gradient_numeric` are `FALSE`.

`.gradient_attribute_name [character(1)]`
: Only relevant if `.gradient_as_attribute = TRUE`.
In that case, the attribute name for the gradient (if available).

`.gradient_numeric [logical(1)]`
: Calculate the gradient via the numerical approximation `grad`?

`.hessian_as_attribute [logical(1)]`
: Add the value of the Hessian function as an attribute to the output?

The attribute name is defined via the `.hessian_attribute_name` argument.
Ignored if `$hessian_specified` and `hessian_numeric` are `FALSE`.

`.hessian_attribute_name [character(1)]`
Only relevant if `.hessian_as_attribute = TRUE`.
In that case, the attribute name for the Hessian (if available).

`.hessian_numeric [logical(1)]`
Calculate the Hessian via the numerical approximation `hessian`?

`...` Optionally additional function arguments that are fixed during the optimization.

Method `evaluate_gradient()`**:** Evaluate the gradient function.

Usage:

```
Objective$evaluate_gradient(.at, .negate = FALSE, ...)
```

Arguments:

`.at [numeric()]`
The values for the target argument(s), written in a single vector.
Must be of length `sum(self$npar)`.

`.negate [logical(1)]`
Negate the function return value?

`...` Optionally additional function arguments that are fixed during the optimization.

Method `evaluate_gradient_numeric()`**:** Evaluate the numerical gradient `grad`.

Usage:

```
Objective$evaluate_gradient_numeric(.at, .negate = FALSE, ...)
```

Arguments:

`.at [numeric()]`
The values for the target argument(s), written in a single vector.
Must be of length `sum(self$npar)`.

`.negate [logical(1)]`
Negate the function return value?

`...` Optionally additional function arguments that are fixed during the optimization.

Method `evaluate_hessian()`**:** Evaluate the Hessian function.

Usage:

```
Objective$evaluate_hessian(.at, .negate = FALSE, ...)
```

Arguments:

`.at [numeric()]`
The values for the target argument(s), written in a single vector.
Must be of length `sum(self$npar)`.

`.negate [logical(1)]`
Negate the function return value?

`...` Optionally additional function arguments that are fixed during the optimization.

Method `evaluate_hessian_numeric()`**:** Evaluate the numerical Hessian `hessian`.

Usage:

```
Objective$evaluate_hessian_numeric(.at, .negate = FALSE, ...)
```

Arguments:

`.at` [`numeric()`]
: The values for the target argument(s), written in a single vector. Must be of length `sum(self$npar)`.

`.negate` [`logical(1)`]
: Negate the function return value?

`...` Optionally additional function arguments that are fixed during the optimization.

Method `print()`**:** Print details of the `Objective` object.

Usage:

```
Objective$print()
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Objective$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
### define log-likelihood function of Gaussian mixture model
llk <- function(mu, sd, lambda, data) {
  sd <- exp(sd)
  lambda <- plogis(lambda)
  cluster_1 <- lambda * dnorm(data, mu[1], sd[1])
  cluster_2 <- (1 - lambda) * dnorm(data, mu[2], sd[2])
  sum(log(cluster_1 + cluster_2))
}

### optimize over the first three arguments, the 'data' argument is constant
objective <- Objective$new(
  f = llk, target = c("mu", "sd", "lambda"), npar = c(2, 2, 1),
  data = faithful$eruptions
)

### evaluate at 1:5 (1:2 is passed to mu, 3:4 to sd, and 5 to lambda)
objective$evaluate(1:5)
```

`Optimizer` *Specify numerical optimizer object*

Description

The `Optimizer` object defines a numerical optimizer based on any optimization algorithm implemented in R. The main advantage of working with an `Optimizer` object instead of using the optimization function directly lies in the standardized inputs and outputs.

Any R function that fulfills the following four constraints can be defined as an `Optimizer` object:

1. It must have an input for a `function`, the objective function to be optimized.
2. It must have an input for a `numeric` vector, the initial values from where the optimizer starts.
3. It must have a `...` argument for additional parameters passed on to the objective function.
4. The output must be a named `list`, including the optimal function value and the optimal parameter vector.

Active bindings

`label` [`character(1)`]
: The label for the optimizer.

`algorithm` [`function`]
: The optimization algorithm.

`arg_objective` [`character(1)`]
: The argument name for the objective function in `algorithm`.

`arg_initial` [`character(1)`]
: The argument name for the initial values in `algorithm`.

`arg_lower` [`character(1)` | `NA`]
: Optionally the argument name for the lower parameter bound in `algorithm`.
Can be `NA` if not available.

`arg_upper` [`character(1)` | `NA`]
: Optionally the argument name for the upper parameter bound in `algorithm`.
Can be `NA` if not available.

`arg_gradient` [`character(1)` | `NA`]
: Optionally the argument name for the gradient function in `algorithm`.
Can be `NA` if not available.

`arg_hessian` [`character(1)` | `NA`]
: Optionally the argument name for the Hessian function in `algorithm`.
Can be `NA` if not available.

`gradient_as_attribute` [`logical(1)`]
: Only relevant if `arg_gradient` is not `NA`.
In that case, does `algorithm` expect that the gradient is an attribute of the objective function output (as for example in `nlm`)? In that case, `arg_gradient` defines the attribute name.

`hessian_as_attribute` [`logical(1)`]
: Only relevant if `arg_hessian` is not `NA`.
In that case, does `algorithm` expect that the Hessian is an attribute of the objective function output (as for example in `nlm`)? In that case, `arg_hessian` defines the attribute name.

`out_value` [`character(1)`]
: The element name for the optimal function value in the output `list` of `algorithm`.

`out_parameter` [`character(1)`]
: The element name for the optimal parameters in the output `list` of `algorithm`.

`direction` [`character(1)`]
: Either `"min"` (if the optimizer minimizes) or `"max"` (if the optimizer maximizes).

`arguments` [`list()`]
: Custom arguments for `algorithm`.

Defaults are used for arguments that are not specified.

`seconds` [`numeric(1)`]
: A time limit in seconds.

Optimization is interrupted prematurely if `seconds` is exceeded.

No time limit if `seconds = Inf` (the default).

Note the limitations documented in `setTimeLimit`.

`hide_warnings` [`logical(1)`]
: Hide warnings during optimization?

`output_ignore` [`character()`]
: Elements to ignore (not include) in the optimization output.

Methods

Public methods:

- `Optimizer$new()`
- `Optimizer$definition()`
- `Optimizer$set_arguments()`
- `Optimizer$minimize()`
- `Optimizer$maximize()`
- `Optimizer$optimize()`
- `Optimizer$print()`
- `Optimizer$clone()`

Method `new()`**:** Initializes a new `Optimizer` object.

Usage:

```
Optimizer$new(which, ..., .verbose = TRUE)
```

Arguments:

`which` [`character(1)`]
: Either:

- one of `optimizer_dictionary$keys`
- or `"custom"`, in which case `$definition()` must be used to define the optimizer details.

`...` [`any`]
: Optionally additional named arguments to be passed to the optimizer algorithm. Without specifications, default values of the optimizer are used.

`.verbose` [`logical(1)`]
: Print status messages?

Returns: A new `Optimizer` object.

Method `definition()`**:** Defines an optimizer.

Usage:

```
Optimizer$definition(
  algorithm,
  arg_objective,
  arg_initial,
  arg_lower = NA,
  arg_upper = NA,
  arg_gradient = NA,
  arg_hessian = NA,
  gradient_as_attribute = FALSE,
  hessian_as_attribute = FALSE,
  out_value,
  out_parameter,
  direction
)
```

Arguments:

`algorithm` [`function`]
: The optimization algorithm.

`arg_objective` [`character(1)`]
: The argument name for the objective function in `algorithm`.

`arg_initial` [`character(1)`]
: The argument name for the initial values in `algorithm`.

`arg_lower` [`character(1)` | `NA`]
: Optionally the argument name for the lower parameter bound in `algorithm`.
Can be `NA` if not available.

`arg_upper` [`character(1)` | `NA`]
: Optionally the argument name for the upper parameter bound in `algorithm`.
Can be `NA` if not available.

`arg_gradient` [`character(1)` | `NA`]
: Optionally the argument name for the gradient function in `algorithm`.
Can be `NA` if not available.

`arg_hessian` [`character(1)` | `NA`]
: Optionally the argument name for the Hessian function in `algorithm`.
Can be `NA` if not available.

`gradient_as_attribute` [`logical(1)`]
: Only relevant if `arg_gradient` is not `NA`.
In that case, does `algorithm` expect that the gradient is an attribute of the objective function output (as for example in `nlm`)? In that case, `arg_gradient` defines the attribute name.

`hessian_as_attribute` [`logical(1)`]
: Only relevant if `arg_hessian` is not `NA`.
In that case, does `algorithm` expect that the Hessian is an attribute of the objective function output (as for example in `nlm`)? In that case, `arg_hessian` defines the attribute name.

`out_value` [`character(1)`]
: The element name for the optimal function value in the output `list` of `algorithm`.

`out_parameter` [`character(1)`]
: The element name for the optimal parameters in the output `list` of `algorithm`.

`direction` [`character(1)`]
Either `"min"` (if the optimizer minimizes) or `"max"` (if the optimizer maximizes).

Method `set_arguments()`**:** Sets optimizer arguments.

Usage:

```
Optimizer$set_arguments(...)
```

Arguments:

`...` [`any`]
Optionally additional named arguments to be passed to the optimizer algorithm. Without specifications, default values of the optimizer are used.

Returns: The `Optimizer` object.

Method `minimize()`**:** Performing minimization.

Usage:

```
Optimizer$minimize(objective, initial, lower = NA, upper = NA, ...)
```

Arguments:

`objective` [`function` | `Objective`]
A `function` to be optimized that

1. has at least one argument that receives a `numeric` `vector`
2. and returns a single `numeric` value.

Alternatively, it can also be an `Objective` object for more flexibility.

`initial` [`numeric()`]
Starting parameter values for the optimization.

`lower` [`NA` | `numeric()` | `numeric(1)`]
Lower bounds on the parameters.
If a single number, this will be applied to all parameters.
Can be `NA` to not define any bounds.

`upper` [`NA` | `numeric()` | `numeric(1)`]
Upper bounds on the parameters.
If a single number, this will be applied to all parameters.
Can be `NA` to not define any bounds.

`...` [`any`]
Optionally additional named arguments to be passed to the optimizer algorithm. Without specifications, default values of the optimizer are used.

Returns: A named `list`, containing at least these five elements:

`value` A `numeric`, the minimum function value.

`parameter` A `numeric` vector, the parameter vector where the minimum is obtained.

`seconds` A `numeric`, the optimization time in seconds.

`initial` A `numeric`, the initial parameter values.

`error` Either `TRUE` if an error occurred, or `FALSE`, else.

Appended are additional output elements of the optimizer.
If an error occurred, then the error message is also appended as element `error_message`.
If the time limit was exceeded, this counts as an error. In addition, the flag `time_out` = `TRUE` is appended.

Examples:

```
Optimizer$new("stats::nlm")$
  minimize(objective = function(x) x^4 + 3*x - 5, initial = 2)
```

Method `maximize()`**:** Performing maximization.

Usage:

```
Optimizer$maximize(objective, initial, lower = NA, upper = NA, ...)
```

Arguments:

`objective` [`function` | `Objective`]
: A `function` to be optimized that

1. has at least one argument that receives a `numeric` `vector`
2. and returns a single `numeric` value.

Alternatively, it can also be an `Objective` object for more flexibility.

`initial` [`numeric()`]
: Starting parameter values for the optimization.

`lower` [`NA` | `numeric()` | `numeric(1)`]
: Lower bounds on the parameters.
If a single number, this will be applied to all parameters.
Can be `NA` to not define any bounds.

`upper` [`NA` | `numeric()` | `numeric(1)`]
: Upper bounds on the parameters.
If a single number, this will be applied to all parameters.
Can be `NA` to not define any bounds.

`...` [`any`]
: Optionally additional named arguments to be passed to the optimizer algorithm. Without specifications, default values of the optimizer are used.

Returns: A named `list`, containing at least these five elements:

`value` A `numeric`, the maximum function value.

`parameter` A `numeric` vector, the parameter vector where the maximum is obtained.

`seconds` A `numeric`, the optimization time in seconds.

`initial` A `numeric`, the initial parameter values.

`error` Either `TRUE` if an error occurred, or `FALSE`, else.

Appended are additional output elements of the optimizer.
If an error occurred, then the error message is also appended as element `error_message`.
If the time limit was exceeded, this also counts as an error. In addition, the flag `time_out = TRUE` is appended.

Examples:

```
Optimizer$new("stats::nlm")$
  maximize(objective = function(x) -x^4 + 3*x - 5, initial = 2)
```

Method `optimize()`**:** Performing minimization or maximization.

Usage:

```
Optimizer$optimize(
  objective,
  initial,
  lower = NA,
  upper = NA,
  direction = "min",
  ...
)
```

Arguments:

`objective` [`function` | `Objective`]
: A `function` to be optimized that

1. has at least one argument that receives a `numeric vector`
2. and returns a single `numeric` value.

Alternatively, it can also be an `Objective` object for more flexibility.

`initial` [`numeric()`]
: Starting parameter values for the optimization.

`lower` [`NA` | `numeric()` | `numeric(1)`]
: Lower bounds on the parameters.
If a single number, this will be applied to all parameters.
Can be `NA` to not define any bounds.

`upper` [`NA` | `numeric()` | `numeric(1)`]
: Upper bounds on the parameters.
If a single number, this will be applied to all parameters.
Can be `NA` to not define any bounds.

`direction` [`character(1)`]
: Either `"min"` for minimization or `"max"` for maximization.

`...` [`any`]
: Optionally additional named arguments to be passed to the optimizer algorithm. Without specifications, default values of the optimizer are used.

Returns: A named `list`, containing at least these five elements:

`value` A `numeric`, the maximum function value.

`parameter` A `numeric` vector, the parameter vector where the maximum is obtained.

`seconds` A `numeric`, the optimization time in seconds.

`initial` A `numeric`, the initial parameter values.

`error` Either `TRUE` if an error occurred, or `FALSE`, else.

Appended are additional output elements of the optimizer.
If an error occurred, then the error message is also appended as element `error_message`.
If the time limit was exceeded, this also counts as an error. In addition, the flag `time_out = TRUE` is appended.

Examples:

```
objective <- function(x) -x^4 + 3*x - 5
optimizer <- Optimizer$new("stats::nlm")
optimizer$optimize(objective = objective, initial = 2, direction = "min")
optimizer$optimize(objective = objective, initial = 2, direction = "max")
```

Method `print()`**:** Prints the optimizer label.

Usage:

```
Optimizer$print(...)
```

Arguments:

`...` [any]
Optionally additional named arguments to be passed to the optimizer algorithm. Without specifications, default values of the optimizer are used.

Returns: Invisibly the `Optimizer` object.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Optimizer$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
### Task: compare minimization with 'stats::nlm' and 'pracma::nelder_mead'

# 1. define objective function and initial values
objective <- TestFunctions::TF_ackley
initial <- c(3, 3)

# 2. get overview of optimizers available in dictionary
optimizer_dictionary$keys

# 3. define 'nlm' optimizer
nlm <- Optimizer$new(which = "stats::nlm")

# 4. define the 'pracma::nelder_mead' optimizer (not contained in the dictionary)
nelder_mead <- Optimizer$new(which = "custom")
nelder_mead$definition(
  algorithm = pracma::nelder_mead, # optimization function
  arg_objective = "fn",            # argument name for the objective function
  arg_initial = "x0",              # argument name for the initial values
  out_value = "fmin",             # element for the optimal function value in the output
  out_parameter = "xmin",          # element for the optimal parameters in the output
  direction = "min"                # optimizer minimizes
)

# 5. compare the minimization results
nlm$minimize(objective, initial)
nelder_mead$minimize(objective, initial)


## ------------------------------------------------
## Method `Optimizer$minimize`
## ------------------------------------------------
```

```
Optimizer$new("stats::nlm")$
  minimize(objective = function(x) x^4 + 3*x - 5, initial = 2)

## ------------------------------------------------
## Method `Optimizer$maximize`
## ------------------------------------------------

Optimizer$new("stats::nlm")$
  maximize(objective = function(x) -x^4 + 3*x - 5, initial = 2)

## ------------------------------------------------
## Method `Optimizer$optimize`
## ------------------------------------------------

objective <- function(x) -x^4 + 3*x - 5
optimizer <- Optimizer$new("stats::nlm")
optimizer$optimize(objective = objective, initial = 2, direction = "min")
optimizer$optimize(objective = objective, initial = 2, direction = "max")
```

`optimizer_dictionary` *Dictionary of optimizer functions*

Description

A dictionary of currently included numerical optimizer functions in the {optimizeR} package.

Usage

```
optimizer_dictionary
```

Format

An R6 object of class Dictionary.

Examples

```
print(optimizer_dictionary)
```

`ParameterSpaces` *Switch between parameter spaces*

Description

The `ParameterSpaces` object manages two related parameter spaces:

- the Optimization Space (for optimization)
- and the Interpretation Space (for easier interpretation).

In the Optimization Space, parameters are stored as a `numeric vector`, the standard format for numerical optimizers.

In the Interpretation Space, parameters are stored as a `list` and can take different formats (e.g., `matrix`).

The user can define transformation functions (not necessarily bijective) to switch between these spaces via the `$o2i()` and `$i2o()` methods.

Methods

Public methods:

- `ParameterSpaces$new()`
- `ParameterSpaces$print()`
- `ParameterSpaces$switch()`
- `ParameterSpaces$o2i()`
- `ParameterSpaces$i2o()`
- `ParameterSpaces$clone()`

Method `new()`**:** Initializes a new `ParameterSpaces` object.

Usage:

```
ParameterSpaces$new(parameter_names, parameter_lengths_in_o_space)
```

Arguments:

`parameter_names [character()]`
Unique names for the parameters.

`parameter_lengths_in_o_space [integer()]`
The length of each parameter in the optimization space.

Returns: A new `ParameterSpaces` object.

Method `print()`**:** Print an overview of the parameter spaces.

Usage:

```
ParameterSpaces$print(show_transformer = FALSE)
```

Arguments:

`show_transformer [logical(1)]`
Show transformer functions in the output?

Method `switch()`**:** Switch between Optimization Space and Interpretation Space.

Usage:

```
ParameterSpaces$switch(x, to = NULL)
```

Arguments:

x [numeric() | list()]
The parameters, either as a numeric vector (will be switched to Interpretation Space), or as a list() (will be switched to Optimization Space).

to [character(1) | NULL]
Explicitly switch to a specific space, either

- "o": Optimization Space
- "i": Interpretation Space

If NULL, the function will switch to the other space.

Method o2i(): Define transformation functions when switching from Optimization Space to Interpretation Space.

Usage:

```
ParameterSpaces$o2i(...)
```

Arguments:

... [function]
One or more transformation functions, named according to the parameters.
Transformers from Optimization Space to Interpretation Space (o2i) **must receive** a numeric.
The default is the identity.

Method i2o(): Define transformation functions when switching from Interpretation Space to Optimization Space.

Usage:

```
ParameterSpaces$i2o(...)
```

Arguments:

... [function]
One or more transformers functions, named according to the parameters.
Transformers from Interpretation Space to Optimization Space (i2o) **must return** a numeric.
The default is as.vector().

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
ParameterSpaces$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Examples

```
### Log-likelihood function of two-class Gaussian mixture model with
### parameter vector `theta` that consists of
### - `mu`, mean vector of length 2
### - `sd`, standard deviation vector of length 2, must be positive
### - `lambda`, class probability of length 1, must be between 0 and 1

normal_mixture_llk <- function(theta, data) {
  mu <- theta[1:2]
  sd <- exp(theta[3:4])
```

```
  lambda <- plogis(theta[5])
  c1 <- lambda * dnorm(data, mu[1], sd[1])
  c2 <- (1 - lambda) * dnorm(data, mu[2], sd[2])
  sum(log(c1 + c2))
}

### define parameter spaces
### - `mu` needs no transformation
### - `sd` needs to be real in optimization space and positive in
###    interpretation space
### - `lambda` needs to be real and of length `1` in optimization space, and
###    a probability vector of length `2` in interpretation space

normal_mixture_spaces <- ParameterSpaces$
  new(
    parameter_names = c("mu", "sd", "lambda"),
    parameter_lengths_in_o_space = c(2, 2, 1)
  )$
  o2i(
    "mu" = function(x) x,
    "sd" = function(x) exp(x),
    "lambda" = function(x) c(plogis(x), 1 - plogis(x))
  )$
  i2o(
    "mu" = function(x) x,
    "sd" = function(x) log(x),
    "lambda" = function(x) qlogis(x[1])
  )

### switch between parameter spaces

par <- list(                            # parameters in interpretation space
  "mu" = c(2, 4),
  "sd" = c(0.5, 1),
  "lambda" = c(0.4, 0.6)
)
(x <- normal_mixture_spaces$switch(par)) # switch to optimization space
normal_mixture_llk(
  theta = x, data = datasets::faithful$eruptions
)
normal_mixture_spaces$switch(x)          # switch back
```

Index