

Package ‘paradox’

October 21, 2020

Type Package

Title Define and Work with Parameter Spaces for Complex Algorithms

Version 0.5.0

Description Define parameter spaces, constraints and dependencies for arbitrary algorithms, to program on such spaces. Also includes statistical designs and random samplers. Objects are implemented as 'R6' classes.

License LGPL-3

URL <https://paradox.mlr-org.com>, <https://github.com/mlr-org/paradox>

BugReports <https://github.com/mlr-org/paradox/issues>

Imports backports, checkmate, data.table, mlr3misc (>= 0.5.0), R6

Suggests knitr, lhs, testthat

Encoding UTF-8

NeedsCompilation no

RoxygenNote 7.1.1

Collate 'Condition.R' 'Design.R' 'NoDefault.R' 'Param.R' 'ParamDbl.R' 'ParamFct.R' 'ParamInt.R' 'ParamLgl.R' 'ParamSet.R' 'ParamSetCollection.R' 'ParamUty.R' 'Sampler.R' 'Sampler1D.R' 'SamplerHierarchical.R' 'SamplerJointIndep.R' 'SamplerUnif.R' 'asserts.R' 'generate_design_grid.R' 'generate_design_lhs.R' 'generate_design_random.R' 'helper.R' 'reexports.R' 'zzz.R'

Author Michel Lang [cre, aut] (<<https://orcid.org/0000-0001-9754-0393>>),
Bernd Bischl [aut] (<<https://orcid.org/0000-0001-6002-6980>>),
Jakob Richter [aut] (<<https://orcid.org/0000-0003-4481-5554>>),
Xudong Sun [aut] (<<https://orcid.org/0000-0003-3269-2307>>),
Martin Binder [aut],
Marc Becker [ctb] (<<https://orcid.org/0000-0002-8115-0400>>)

Maintainer Michel Lang <michellang@gmail.com>

Repository CRAN

Date/Publication 2020-10-21 20:30:02 UTC

R topics documented:

paradox-package	2
assert_param	3
Condition	4
Design	5
generate_design_grid	7
generate_design_lhs	8
generate_design_random	9
NO_DEF	9
Param	10
ParamDbf	13
ParamFct	15
ParamInt	17
ParamLgl	19
ParamSet	20
ParamSetCollection	26
ParamUty	28
Sampler	30
Sampler1D	31
Sampler1DCateg	32
Sampler1DNormal	33
Sampler1DRfun	34
Sampler1DUnif	35
SamplerHierarchical	36
SamplerJointIndep	37
SamplerUnif	38
transpose	39
Index	40

paradox-package	<i>paradox: Define and Work with Parameter Spaces for Complex Algorithms</i>
-----------------	--

Description

Define parameter spaces, constraints and dependencies for arbitrary algorithms, to program on such spaces. Also includes statistical designs and random samplers. Objects are implemented as 'R6' classes.

Author(s)

Maintainer: Michel Lang <michellang@gmail.com> ([ORCID](#))

Authors:

- Bernd Bischl <bernd_bischl@gmx.net> ([ORCID](#))
- Jakob Richter <jakob1richter@gmail.com> ([ORCID](#))

- Xudong Sun <smilesun.east@gmail.com> ([ORCID](#))
- Martin Binder <mlr.developer@mb706.com>

Other contributors:

- Marc Becker <marcbecker@posteo.de> ([ORCID](#)) [contributor]

See Also

Useful links:

- <https://paradox.mlr-org.com>
- <https://github.com/mlr-org/paradox>
- Report bugs at <https://github.com/mlr-org/paradox/issues>

assert_param

Assertions for Params and ParamSets

Description

Assertions for Params and ParamSets

Usage

```
assert_param(param, cl = "Param", no_untyped = FALSE, must_bounded = FALSE)
```

```
assert_param_set(
  param_set,
  cl = "Param",
  no_untyped = FALSE,
  must_bounded = FALSE,
  no_deps = FALSE
)
```

Arguments

param	(Param).
cl	(<code>character()</code>) Allowed subclasses.
no_untyped	(<code>logical(1)</code>) Are untyped Params allowed?
must_bounded	(<code>logical(1)</code>) Only bounded Params allowed?
param_set	(ParamSet).
no_deps	(<code>logical(1)</code>) Are dependencies allowed?

Value

The checked object, invisibly.

Condition

Dependency Condition

Description

Condition object, to specify the condition in a dependency.

Currently implemented simple conditions

- `CondEqual$new(rhs)`
Parent must be equal to rhs.
- `CondAnyOf$new(rhs)`
Parent must be any value of rhs.

Public fields

`type` (character(1))
Name / type of the condition.

`rhs` (any)
Right-hand-side of the condition.

Methods**Public methods:**

- `Condition$new()`
- `Condition$test()`
- `Condition$as_string()`
- `Condition$format()`
- `Condition$print()`
- `Condition$clone()`

Method `new()`: Creates a new instance of this R6 class.

Usage:

`Condition$new(type, rhs)`

Arguments:

`type` (character(1))
Name / type of the condition.

`rhs` (any)
Right-hand-side of the condition.

Method `test()`: Checks if condition is satisfied. Called on a vector of parent param values.

Usage:

Condition\$test(x)

Arguments:

x (vector()).

Returns: logical(1).

Method as_string(): Conversion helper for print outputs.

Usage:

Condition\$as_string(lhs_chr = "x")

Arguments:

lhs_chr (character(1))

Method format(): Helper for print outputs.

Usage:

Condition\$format()

Method print(): Printer.

Usage:

Condition\$print(...)

Arguments:

... (ignored).

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

Usage:

Condition\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Design

Design of Configurations

Description

A lightweight wrapper around a [ParamSet](#) and a [data.table::data.table\(\)](#), where the latter is a design of configurations produced from the former - e.g., by calling a [generate_design_grid\(\)](#) or by sampling.

Public fields

param_set ([ParamSet](#)).

data ([data.table::data.table\(\)](#))

Stored data.

Methods

Public methods:

- [Design\\$new\(\)](#)
- [Design\\$format\(\)](#)
- [Design\\$print\(\)](#)
- [Design\\$transpose\(\)](#)
- [Design\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

```
Design$new(param_set, data, remove_dupl)
```

Arguments:

`param_set` ([ParamSet](#)).

`data` ([data.table::data.table\(\)](#))

Stored data.

`remove_dupl` (`logical(1)`)

Remove duplicates?

Method `format()`: Helper for print outputs.

Usage:

```
Design$format()
```

Method `print()`: Printer.

Usage:

```
Design$print(...)
```

Arguments:

... (ignored).

Method `transpose()`: Converts data into a list of lists of row-configurations, possibly removes NA entries of inactive parameter values due to unsatisfied dependencies, and possibly calls the `trafo` function of the [ParamSet](#).

Usage:

```
Design$transpose(filter_na = TRUE, trafo = TRUE)
```

Arguments:

`filter_na` (`logical(1)`)

Should NA entries of inactive parameter values due to unsatisfied dependencies be removed?

`trafo` (`logical(1)`)

Should the `trafo` function of the [ParamSet](#) be called?

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

generate_design_grid *Generate a Grid Design*

Description

Generate a grid with a specified resolution in the parameter space. The resolution for categorical parameters is ignored, these parameters always produce a grid over all their valid levels. For number params the endpoints of the params are always included in the grid.

Usage

```
generate_design_grid(param_set, resolution = NULL, param_resolutions = NULL)
```

Arguments

param_set ([ParamSet](#)).

resolution ([integer\(1\)](#))
 Global resolution for all [Params](#).

param_resolutions
 ([named integer\(\)](#))
 Resolution per [Param](#), named by parameter ID.

Value

[Design](#).

See Also

Other generate_design: [generate_design_lhs\(\)](#), [generate_design_random\(\)](#)

Examples

```
ps = ParamSet$new(list(  
  ParamDbf$new("ratio", lower = 0, upper = 1),  
  ParamFct$new("letters", levels = letters[1:3])  
)  
)  
generate_design_grid(ps, 10)
```

generate_design_lhs *Generate a Space-Filling LHS Design*

Description

Generate a space-filling design using Latin hypercube sampling.

Usage

```
generate_design_lhs(param_set, n, lhs_fun = NULL)
```

Arguments

param_set	(ParamSet).
n	(integer(1)) Number of points to sample.
lhs_fun	(function(n, k)) Function to use to generate a LHS sample, with n samples and k values per param. LHS functions are implemented in package lhs , default is to use lhs::maximinLHS() .

Value

[Design](#).

See Also

Other generate_design: [generate_design_grid\(\)](#), [generate_design_random\(\)](#)

Examples

```
ps = ParamSet$new(list(
  ParamDbf$new("ratio", lower = 0, upper = 1),
  ParamFct$new("letters", levels = letters[1:3])
))

if (requireNamespace("lhs", quietly = TRUE)) {
  generate_design_lhs(ps, 10)
}
```

`generate_design_random`*Generate a Random Design*

Description

Generates a design with randomly drawn points. Internally uses [SamplerUnif](#), hence, also works for [ParamSets](#) with dependencies. If dependencies do not hold, values are set to NA in the resulting `data.table`.

Usage

```
generate_design_random(param_set, n)
```

Arguments

<code>param_set</code>	(ParamSet).
<code>n</code>	(<code>integer(1)</code>) Number of points to draw randomly.

Value

[Design](#).

See Also

Other `generate_design`: [generate_design_grid\(\)](#), [generate_design_lhs\(\)](#)

Examples

```
ps = ParamSet$new(list(
  ParamDbl$new("ratio", lower = 0, upper = 1),
  ParamFct$new("letters", levels = letters[1:3])
))
generate_design_random(ps, 10)
```

`NO_DEF`*Extra data type for "no default value"*

Description

Special new data type for no-default. Not often needed by the end-user, mainly internal.

- `NoDefault`: R6 factory.
- `NO_DEF`: R6 Singleton object for type, used in [Param](#).
- `is_nodefult()`: Is an object of type 'no default'?

Param

*Param Class***Description**

This is the abstract base class for parameter objects like [ParamDbl](#) and [ParamFct](#).

S3 methods

- `as.data.table()`
[Param](#) -> `data.table::data.table()`
 Converts param to `data.table::data.table()` with 1 row. See [ParamSet](#).

Public fields

`id` (character(1))
 Identifier of the object.

`special_vals` (list())
 Arbitrary special values this parameter is allowed to take.

`default` (any)
 Default value.

`tags` (character())
 Arbitrary tags to group and subset parameters.

Active bindings

`class` (character(1))
 R6 class name. Read-only.

`is_number` (logical(1))
 TRUE if the parameter is of type "dbl" or "int".

`is_categ` (logical(1))
 TRUE if the parameter is of type "fct" or "lgl".

`has_default` (logical(1))
 Is there a default value?

Methods**Public methods:**

- [Param\\$new\(\)](#)
- [Param\\$check\(\)](#)
- [Param\\$assert\(\)](#)
- [Param\\$test\(\)](#)
- [Param\\$rep\(\)](#)
- [Param\\$format\(\)](#)

- [Param\\$print\(\)](#)
- [Param\\$qunif\(\)](#)
- [Param\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Note that this object is typically constructed via derived classes, e.g., [ParamDbf](#).

Usage:

```
Param$new(id, special_vals, default, tags)
```

Arguments:

`id` (`character(1)`)

Identifier of the object.

`special_vals` (`list()`)

Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.

`default` (`any`)

Default value. Can be from the domain of the parameter or an element of `special_vals`. Has value `NO_DEF` if no default exists. `NULL` can be a valid default. The value has no effect on `ParamSet$values` or the behavior of `ParamSet$check()`, `$test()` or `$assert()`. The default is intended to be used for documentation purposes. ‘

`tags` (`character()`)

Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [Param-Set](#).

Method `check()`: **checkmate**-like check-function. Take a value from the domain of the parameter, and check if it is feasible. A value is feasible if it is of the same `storage_type`, inside of the bounds or element of `special_vals`.

Usage:

```
Param$check(x)
```

Arguments:

`x` (`any`).

Returns: If successful `TRUE`, if not a string with the error message.

Method `assert()`: **checkmate**-like assert-function. Take a value from the domain of the parameter, and assert if it is feasible. A value is feasible if it is of the same `storage_type`, inside of the bounds or element of `special_vals`.

Usage:

```
Param$assert(x)
```

Arguments:

`x` (`any`).

Returns: If successful `x` invisibly, if not an exception is raised.

Method test(): **checkmate**-like test-function. Take a value from the domain of the parameter, and test if it is feasible. A value is feasible if it is of the same `storage_type`, inside of the bounds or element of `special_vals`.

Usage:

`Param$test(x)`

Arguments:

`x` (any).

Returns: If successful TRUE, if not FALSE.

Method rep(): Repeats this parameter `n`-times (by cloning). Each parameter is named "`[id]rep[k]`" and gets the additional tag "`[id]_rep`".

Usage:

`Param$rep(n)`

Arguments:

`n` (`integer(1)`).

Returns: [ParamSet](#).

Method format(): Helper for print outputs.

Usage:

`Param$format()`

Method print(): Printer.

Usage:

`Param$print(`

```

    ...,
    hide_cols = c("nlevels", "is_bounded", "special_vals", "tags", "storage_type")
)
```

Arguments:

`...` (ignored).

`hide_cols` (`character()`)

Which fields should not be printed? Default is "nlevels", "is_bounded", "special_vals", "tags", and "storage_type".

Method qunif(): Takes values from `[0,1]` and maps them, regularly distributed, to the domain of the parameter. Think of: quantile function or the use case to map a uniform-`[0,1]` random variable into a uniform sample from this param.

Usage:

`Param$qunif(x)`

Arguments:

`x` (`numeric(1)`).

Returns: Value of the domain of the parameter.

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

Usage:

`Param$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Params: [ParamDb1](#), [ParamFct](#), [ParamInt](#), [ParamLgl](#), [ParamUty](#)

ParamDb1	<i>Numerical Parameter</i>
----------	----------------------------

Description

A [Param](#) to describe real-valued parameters.

Super class

`paradox::Param -> ParamDb1`

Public fields

`lower` (numeric(1))
Lower bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

`upper` (numeric(1))
Upper bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

Active bindings

`levels` (character() | NULL)
Set of allowed levels. Always NULL for [ParamDb1](#), [ParamInt](#) and [ParamUty](#). Always `c(TRUE, FALSE)` for [ParamLgl](#).

`nlevels` (integer(1) | Inf)
Number of categorical levels. Always Inf for [ParamDb1](#) and [ParamUty](#). The number of integers in the range [lower, upper], or Inf if unbounded for [ParamInt](#). Always 2 for [ParamLgl](#).

`is_bounded` (logical(1))
Are the bounds finite? Always TRUE for [ParamFct](#) and [ParamLgl](#). Always FALSE for [ParamUty](#).

`storage_type` (character(1))
Data type when values of this parameter are stored in a data table or sampled. Always "numeric" for [ParamDb1](#). Always "character" for [ParamFct](#). Always "integer" for [ParamInt](#). Always "logical" for [ParamLgl](#). Always "list" for [ParamUty](#).

Methods**Public methods:**

- [ParamDb1\\$new\(\)](#)
- [ParamDb1\\$clone\(\)](#)

Method `new()`: Creates a new instance of this R6 class.

Usage:

```
ParamDb1$new(
  id,
  lower = -Inf,
  upper = Inf,
  special_vals = list(),
  default = NO_DEF,
  tags = character()
)
```

Arguments:

id (character(1))

Identifier of the object.

lower (numeric(1))

Lower bound, can be -Inf.

upper (numeric(1))

Upper bound can be +Inf.

special_vals (list())

Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.

default (any)

Default value. Can be from the domain of the parameter or an element of special_vals. Has value `NO_DEF` if no default exists. NULL can be a valid default. The value has no effect on `ParamSet$values` or the behavior of `ParamSet$check()`, `$test()` or `$assert()`. The default is intended to be used for documentation purposes. ‘

tags (character())

Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [Param-Set](#).

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

Usage:

```
ParamDb1$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Note

The upper and lower bounds in `$check()` are expanded by `sqrt(.Machine$double.eps)` to prevent errors due to the precision of double values.

See Also

Other Params: [ParamFct](#), [ParamInt](#), [ParamLg1](#), [ParamUty](#), [Param](#)

Examples

```
ParamDb1$new("ratio", lower = 0, upper = 1, default = 0.5)
```

ParamFct	<i>Factor Parameter</i>
----------	-------------------------

Description

A [Param](#) to describe categorical (factor) parameters.

Super class

`paradox::Param -> ParamFct`

Public fields

`levels` (character() | NULL)
 Set of allowed levels. Always NULL for [ParamDbl](#), [ParamInt](#) and [ParamUty](#). Always `c(TRUE, FALSE)` for [ParamLgl](#).

Active bindings

`lower` (numeric(1))
 Lower bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

`upper` (numeric(1))
 Upper bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

`nlevels` (integer(1) | Inf)
 Number of categorical levels. Always Inf for [ParamDbl](#) and [ParamUty](#). The number of integers in the range [lower, upper], or Inf if unbounded for [ParamInt](#). Always 2 for [ParamLgl](#).

`is_bounded` (logical(1))
 Are the bounds finite? Always TRUE for [ParamFct](#) and [ParamLgl](#). Always FALSE for [ParamUty](#).

`storage_type` (character(1))
 Data type when values of this parameter are stored in a data table or sampled. Always "numeric" for [ParamDbl](#). Always "character" for [ParamFct](#). Always "integer" for [ParamInt](#). Always "logical" for [ParamLgl](#). Always "list" for [ParamUty](#).

Methods

Public methods:

- [ParamFct\\$new\(\)](#)
- [ParamFct\\$clone\(\)](#)

Method `new()`: Creates a new instance of this R6 class.

Usage:

```
ParamFct$new(
  id,
  levels,
  special_vals = list(),
  default = NO_DEF,
  tags = character()
)
```

Arguments:

`id` (character(1))

Identifier of the object.

`levels` (character())

Set of allowed levels.

`special_vals` (list())

Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.

`default` (any)

Default value. Can be from the domain of the parameter or an element of `special_vals`. Has value `NO_DEF` if no default exists. `NULL` can be a valid default. The value has no effect on `ParamSet$values` or the behavior of `ParamSet$check()`, `$test()` or `$assert()`. The default is intended to be used for documentation purposes. ‘

`tags` (character())

Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [ParamSet](#).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Params: [ParamDb1](#), [ParamInt](#), [ParamLg1](#), [ParamUty](#), [Param](#)

Examples

```
ParamFct$new("f", levels = letters[1:3])
```

ParamInt	<i>Integer Parameter</i>
----------	--------------------------

Description

A [Param](#) to describe integer parameters.

Methods

See [Param](#).

Super class

[paradox::Param](#) -> ParamInt

Public fields

lower (numeric(1))
 Lower bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

upper (numeric(1))
 Upper bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

Active bindings

levels (character() | NULL)
 Set of allowed levels. Always NULL for [ParamDbf](#), [ParamInt](#) and [ParamUty](#). Always c(TRUE, FALSE) for [ParamLgl](#).

nlevels (integer(1) | Inf)
 Number of categorical levels. Always Inf for [ParamDbf](#) and [ParamUty](#). The number of integers in the range [lower, upper], or Inf if unbounded for [ParamInt](#). Always 2 for [ParamLgl](#).

is_bounded (logical(1))
 Are the bounds finite? Always TRUE for [ParamFct](#) and [ParamLgl](#). Always FALSE for [ParamUty](#).

storage_type (character(1))
 Data type when values of this parameter are stored in a data table or sampled. Always "numeric" for [ParamDbf](#). Always "character" for [ParamFct](#). Always "integer" for [ParamInt](#). Always "logical" for [ParamLgl](#). Always "list" for [ParamUty](#).

Methods**Public methods:**

- [ParamInt\\$new\(\)](#)
- [ParamInt\\$clone\(\)](#)

Method [new\(\)](#): Creates a new instance of this R6 class.

Usage:

```

ParamInt$new(
  id,
  lower = -Inf,
  upper = Inf,
  special_vals = list(),
  default = NO_DEF,
  tags = character()
)

```

Arguments:

id (character(1))

Identifier of the object.

lower (numeric(1))

Lower bound, can be -Inf.

upper (numeric(1))

Upper bound can be +Inf.

special_vals (list())

Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.

default (any)

Default value. Can be from the domain of the parameter or an element of special_vals. Has value `NO_DEF` if no default exists. NULL can be a valid default. The value has no effect on `ParamSet$values` or the behavior of `ParamSet$check()`, `$test()` or `$assert()`. The default is intended to be used for documentation purposes. ‘

tags (character())

Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [Param-Set](#).

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Other Params: [ParamDb1](#), [ParamFct](#), [ParamLg1](#), [ParamUty](#), [Param](#)

Examples

```
ParamInt$new("count", lower = 0, upper = 10, default = 1)
```

ParamLgl	<i>Logical Parameter</i>
----------	--------------------------

Description

A [Param](#) to describe logical parameters.

Super class

```
paradox::Param -> ParamLgl
```

Active bindings

`lower` (numeric(1))
Lower bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

`upper` (numeric(1))
Upper bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

`levels` (character() | NULL)
Set of allowed levels. Always NULL for [ParamDbf](#), [ParamInt](#) and [ParamUty](#). Always c(TRUE, FALSE) for [ParamLgl](#).

`nlevels` (integer(1) | Inf)
Number of categorical levels. Always Inf for [ParamDbf](#) and [ParamUty](#). The number of integers in the range [lower, upper], or Inf if unbounded for [ParamInt](#). Always 2 for [ParamLgl](#).

`is_bounded` (logical(1))
Are the bounds finite? Always TRUE for [ParamFct](#) and [ParamLgl](#). Always FALSE for [ParamUty](#).

`storage_type` (character(1))
Data type when values of this parameter are stored in a data table or sampled. Always "numeric" for [ParamDbf](#). Always "character" for [ParamFct](#). Always "integer" for [ParamInt](#). Always "logical" for [ParamLgl](#). Always "list" for [ParamUty](#).

Methods

Public methods:

- [ParamLgl\\$new\(\)](#)
- [ParamLgl\\$clone\(\)](#)

Method `new()`: Creates a new instance of this R6 class.

Usage:

```
ParamLgl$new(id, special_vals = list(), default = NO_DEF, tags = character())
```

Arguments:

`id` (character(1))
Identifier of the object.

`special_vals (list())`

Arbitrary special values this parameter is allowed to take, to make it feasible. This allows extending the domain of the parameter. Note that these values are only used in feasibility checks, neither in generating designs nor sampling.

`default (any)`

Default value. Can be from the domain of the parameter or an element of `special_vals`. Has value `NO_DEF` if no default exists. `NULL` can be a valid default. The value has no effect on `ParamSet$values` or the behavior of `ParamSet$check()`, `$test()` or `$assert()`. The default is intended to be used for documentation purposes. ‘

`tags (character())`

Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [ParamSet](#).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Params: [ParamDb1](#), [ParamFct](#), [ParamInt](#), [ParamUty](#), [Param](#)

Examples

```
ParamLgl$new("flag", default = TRUE)
```

ParamSet

ParamSet

Description

A set of [Param](#) objects. Please note that when creating a set or adding to it, the parameters of the resulting set have to be uniquely named with IDs with valid R names. The set also contains a member variable values which can be used to store an active configuration / or to partially fix some parameters to constant values (regarding subsequent sampling or generation of designs).

S3 methods and type converters

- `as.data.table()`
[ParamSet](#) -> `data.table::data.table()`
 Compact representation as datatable. Col types are:

– `id`: character

- lower, upper: double
- levels: list col, with NULL elements
- special_vals: list col of list
- is_bounded: logical
- default: list col, with NULL elements
- storage_type: character
- tags: list col of character vectors

Public fields

`assert_values` (logical(1))
Should values be checked for validity during assignment to active binding \$values? Default is TRUE, only switch this off if you know what you are doing.

Active bindings

`params` (named list())
List of [Param](#), named with their respective ID.

`deps` (`data.table::data.table()`)
Table has cols `id` (character(1)) and `on` (character(1)) and `cond` ([Condition](#)). Lists all (direct) dependency parents of a param, through parameter IDs. Internally created by a call to `add_dep`. Settable, if you want to remove dependencies or perform other changes.

`set_id` (character(1))
ID of this param set. Default "". Settable.

`length` (integer(1))
Number of contained [Params](#).

`is_empty` (logical(1))
Is the ParamSet empty?

`class` (named character())
Classes of contained parameters, named with parameter IDs.

`lower` (named double())
Lower bounds of parameters (NA if parameter is not numeric). Named with parameter IDs.

`upper` (named double())
Upper bounds of parameters (NA if parameter is not numeric). Named with parameter IDs.

`levels` (named list())
List of character vectors of allowed categorical values of contained parameters. NULL if the parameter is not categorical. Named with parameter IDs.

`nlevels` (named integer())
Number of categorical levels per parameter, Inf for double parameters or unbounded integer parameters. Named with param IDs.

`is_bounded` (named logical())
Do all parameters have finite bounds? Named with parameter IDs.

`special_vals` (named list() of list())
Special values for all parameters. Named with parameter IDs.

`default` (named `list()`)
 Default values of all parameters. If no default exists, element is not present. Named with parameter IDs.

`tags` (named `list()` of `character()`)
 Can be used to group and subset parameters. Named with parameter IDs.

`storage_type` (`character()`)
 Data types of parameters when stored in tables. Named with parameter IDs.

`is_number` (named `logical()`)
 Position is TRUE for [ParamDbf](#) and [ParamInt](#). Named with parameter IDs.

`is_categ` (named `logical()`)
 Position is TRUE for [ParamFct](#) and [ParamLgl](#). Named with parameter IDs.

`is_numeric` (`logical(1)`)
 Is TRUE if all parameters are [ParamDbf](#) or [ParamInt](#).

`is_categorical` (`logical(1)`)
 Is TRUE if all parameters are [ParamFct](#) and [ParamLgl](#).

`trafo` (`function(x, param_set)`)
 Transformation function. Settable. User has to pass a `function(x, param_set)`, of the form `(named list(), ParamSet) -> named list()`.
 The function is responsible to transform a feasible configuration into another encoding, before potentially evaluating the configuration with the target algorithm. For the output, not many things have to hold. It needs to have unique names, and the target algorithm has to accept the configuration. For convenience, the self-paramset is also passed in, if you need some info from it (e.g. tags). Is NULL by default, and you can set it to NULL to switch the transformation off.

`has_trafo` (`logical(1)`)
 Has the set a trafo function?

`values` (named `list()`)
 Currently set / fixed parameter values. Settable, and feasibility of values will be checked when you set them. You do not have to set values for all parameters, but only for a subset. When you set values, all previously set values will be unset / removed.

`has_deps` (`logical(1)`)
 Has the set parameter dependencies?

Methods

Public methods:

- [ParamSet\\$new\(\)](#)
- [ParamSet\\$add\(\)](#)
- [ParamSet\\$ids\(\)](#)
- [ParamSet\\$get_values\(\)](#)
- [ParamSet\\$subset\(\)](#)
- [ParamSet\\$check\(\)](#)
- [ParamSet\\$test\(\)](#)
- [ParamSet\\$assert\(\)](#)

- `ParamSet$check_dt()`
- `ParamSet$test_dt()`
- `ParamSet$assert_dt()`
- `ParamSet$add_dep()`
- `ParamSet$format()`
- `ParamSet$print()`
- `ParamSet$clone()`

Method `new()`: Creates a new instance of this R6 class.

Usage:

```
ParamSet$new(params = named_list())
```

Arguments:

`params` (`list()`)

List of `Param`, named with their respective ID. Parameters are cloned.

Method `add()`: Adds a single param or another set to this set, all params are cloned.

Usage:

```
ParamSet$add(p)
```

Arguments:

`p` (`Param` | `ParamSet`).

Method `ids()`: Retrieves IDs of contained parameters based on some filter criteria selections, NULL means no restriction. Only returns IDs of parameters that satisfy all conditions.

Usage:

```
ParamSet$ids(class = NULL, is_bounded = NULL, tags = NULL)
```

Arguments:

`class` (`character()`).

`is_bounded` (`logical(1)`).

`tags` (`character()`).

Returns: `character()`.

Method `get_values()`: Retrieves parameter values based on some selections, NULL means no restriction and is equivalent to `$values`. Only returns values of parameters that satisfy all conditions.

Usage:

```
ParamSet$get_values(class = NULL, is_bounded = NULL, tags = NULL)
```

Arguments:

`class` (`character()`).

`is_bounded` (`logical(1)`).

`tags` (`character()`).

Returns: `Named list()`.

Method `subset()`: Changes the current set to the set of passed IDs.

Usage:

ParamSet\$subset(ids)

Arguments:

ids (character()).

Method check(): **checkmate**-like check-function. Takes a named list. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should not be part of x.

Usage:

ParamSet\$check(xs)

Arguments:

xs (named list()).

Returns: If successful TRUE, if not a string with the error message.

Method test(): **checkmate**-like test-function. Takes a named list. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should not be part of x.

Usage:

ParamSet\$test(xs)

Arguments:

xs (named list()).

Returns: If successful TRUE, if not FALSE.

Method assert(): **checkmate**-like assert-function. Takes a named list. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should not be part of x.

Usage:

ParamSet\$assert(xs, .var.name = vname(xs))

Arguments:

xs (named list()).

.var.name (character(1))

Name of the checked object to print in error messages.

Defaults to the heuristic implemented in [vname](#).

Returns: If successful xs invisibly, if not an error message.

Method check_dt(): **checkmate**-like check-function. Takes a [data.table::data.table](#) where rows are points and columns are parameters. A point x is feasible, if it configures a subset of params, all individual param constraints are satisfied and all dependencies are satisfied. Params for which dependencies are not satisfied should be set to NA in xdt.

Usage:

ParamSet\$check_dt(xdt)

Arguments:

xdt ([data.table::data.table](#)).

Returns: If successful TRUE, if not a string with the error message.

Method `test_dt()`: **checkmate**-like test-function (s. `$check_dt()`).

Usage:

```
ParamSet$test_dt(xdt)
```

Arguments:

`xdt` ([data.table::data.table](#)).

Returns: If successful TRUE, if not FALSE.

Method `assert_dt()`: **checkmate**-like assert-function (s. `$check_dt()`).

Usage:

```
ParamSet$assert_dt(xdt, .var.name = vname(xdt))
```

Arguments:

`xdt` ([data.table::data.table](#)).

`.var.name` (`character(1)`)

Name of the checked object to print in error messages.

Defaults to the heuristic implemented in [vname](#).

Returns: If successful `xs` invisibly, if not an error message.

Method `add_dep()`: Adds a dependency to this set, so that param `id` now depends on param `on`.

Usage:

```
ParamSet$add_dep(id, on, cond)
```

Arguments:

`id` (`character(1)`).

`on` (`character(1)`).

`cond` ([Condition](#)).

Method `format()`: Helper for print outputs.

Usage:

```
ParamSet$format()
```

Method `print()`: Printer.

Usage:

```
ParamSet$print(
  ...,
  hide_cols = c("nlevels", "is_bounded", "special_vals", "tags", "storage_type")
)
```

Arguments:

... (ignored).

`hide_cols` (`character()`)

Which fields should not be printed? Default is "nlevels", "is_bounded", "special_vals", "tags", and "storage_type".

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```

ps = ParamSet$new(
  params = list(
    ParamDbf$new("d", lower = -5, upper = 5, default = 0),
    ParamFct$new("f", levels = letters[1:3])
  )
)

ps$trafo = function(x, param_set) {
  x$d = 2^x$d
  return(x)
}

ps$add(ParamInt$new("i", lower = 0L, upper = 16L))

ps$check(list(d = 2.1, f = "a", i = 3L))

```

ParamSetCollection	<i>ParamSetCollection</i>
--------------------	---------------------------

Description

A collection of multiple [ParamSet](#) objects.

- The collection is basically a light-weight wrapper / container around references to multiple sets.
- In order to ensure unique param names, every param in the collection is referred to with "`<set_id>.<param_id>`". Parameters from ParamSets with empty (i.e. `""`) `$set_id` are referenced directly. Multiple ParamSets with `$set_id ""` can be combined, but their parameter names must be unique.
- Operation subset is currently not allowed.
- Operation add currently only works when adding complete sets not single params.
- When you either ask for 'values' or set them, the operation is delegated to the individual, contained param set references. The collection itself does not maintain a values state. This also implies that if you directly change values in one of the referenced sets, this change is reflected in the collection.
- Dependencies: It is possible to currently handle dependencies
 - regarding parameters inside of the same set - in this case simply add the dependency to the set, best before adding the set to the collection
 - across sets, where a param from one set depends on the state of a param from another set - in this case add call `add_dep` on the collection.

If you call `deps` on the collection, you are returned a complete table of dependencies, from sets and across sets.

Super class

`paradox::ParamSet` -> ParamSetCollection

Active bindings

params (named list())

List of [Param](#), named with their respective ID.

deps ([data.table::data.table\(\)](#))

Table has cols id (character(1)) and on (character(1)) and cond ([Condition](#)). Lists all (direct) dependency parents of a param, through parameter IDs. Internally created by a call to `add_dep`. Settable, if you want to remove dependencies or perform other changes.

values (named list())

Currently set / fixed parameter values. Settable, and feasibility of values will be checked when you set them. You do not have to set values for all parameters, but only for a subset. When you set values, all previously set values will be unset / removed.

Methods**Public methods:**

- [ParamSetCollection\\$new\(\)](#)
- [ParamSetCollection\\$add\(\)](#)
- [ParamSetCollection\\$remove_sets\(\)](#)
- [ParamSetCollection\\$subset\(\)](#)
- [ParamSetCollection\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

`ParamSetCollection$new(sets)`

Arguments:

sets (list() of [ParamSet](#))

Parameter objects are cloned.

Method `add()`: Adds a set to this collection.

Usage:

`ParamSetCollection$add(p)`

Arguments:

p ([ParamSet](#)).

Method `remove_sets()`: Removes sets of given ids from collection.

Usage:

`ParamSetCollection$remove_sets(ids)`

Arguments:

ids (character()).

Method `subset()`: Only included for consistency. Not allowed to perform on [ParamSetCollections](#).

Usage:

`ParamSetCollection$subset(ids)`

Arguments:

ids (character()).

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

ParamUty

Untyped Parameter

Description

A [Param](#) to describe untyped parameters.

Super class

```
paradox: :Param -> ParamUty
```

Public fields

custom_check (function())
Custom function to check the feasibility.

Active bindings

lower (numeric(1))
Lower bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

upper (numeric(1))
Upper bound. Always NA for [ParamFct](#), [ParamLgl](#) and [ParamUty](#).

levels (character() | NULL)
Set of allowed levels. Always NULL for [ParamDbf](#), [ParamInt](#) and [ParamUty](#). Always c(TRUE, FALSE) for [ParamLgl](#).

nlevels (integer(1) | Inf)
Number of categorical levels. Always Inf for [ParamDbf](#) and [ParamUty](#). The number of integers in the range [lower, upper], or Inf if unbounded for [ParamInt](#). Always 2 for [ParamLgl](#).

is_bounded (logical(1))
Are the bounds finite? Always TRUE for [ParamFct](#) and [ParamLgl](#). Always FALSE for [ParamUty](#).

storage_type (character(1))
Data type when values of this parameter are stored in a data table or sampled. Always "numeric" for [ParamDbf](#). Always "character" for [ParamFct](#). Always "integer" for [ParamInt](#). Always "logical" for [ParamLgl](#). Always "list" for [ParamUty](#).

Methods

Public methods:

- [ParamUty\\$new\(\)](#)
- [ParamUty\\$clone\(\)](#)

Method new(): Creates a new instance of this [R6](#) class.

Usage:

```
ParamUty$new(id, default = NO_DEF, tags = character(), custom_check = NULL)
```

Arguments:

id (character(1))

Identifier of the object.

default (any)

Default value. Can be from the domain of the parameter or an element of `special_vals`. Has value `NO_DEF` if no default exists. `NULL` can be a valid default. The value has no effect on `ParamSet$values` or the behavior of `ParamSet$check()`, `$test()` or `$assert()`. The default is intended to be used for documentation purposes. ‘

tags (character())

Arbitrary tags to group and subset parameters. Some tags serve a special purpose:

- "required" implies that the parameters has to be given when setting values in [ParamSet](#).

custom_check (function())

Custom function to check the feasibility. Function which checks the input. Must return 'TRUE' if the input is valid and a string with the error message otherwise. Defaults to `NULL`, which means that no check is performed.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Other Params: [ParamDbl](#), [ParamFct](#), [ParamInt](#), [ParamLgl](#), [Param](#)

Examples

```
ParamUty$new("untyped", default = Inf)
```

Sampler

Sampler Class

Description

This is the abstract base class for sampling objects like [Sampler1D](#), [SamplerHierarchical](#) or [SamplerJointIndep](#).

Public fields

param_set ([ParamSet](#))

Domain / support of the distribution we want to sample from.

Methods

Public methods:

- [Sampler\\$new\(\)](#)
- [Sampler\\$sample\(\)](#)
- [Sampler\\$format\(\)](#)
- [Sampler\\$print\(\)](#)
- [Sampler\\$clone\(\)](#)

Method `new()`: Creates a new instance of this R6 class.

Note that this object is typically constructed via derived classes, e.g., [Sampler1D](#).

Usage:

```
Sampler$new(param_set)
```

Arguments:

param_set ([ParamSet](#))

Domain / support of the distribution we want to sample from. ParamSet is cloned on construction.

Method `sample()`: Sample `n` values from the distribution.

Usage:

```
Sampler$sample(n)
```

Arguments:

`n` (`integer(1)`).

Returns: [Design](#).

Method `format()`: Helper for print outputs.

Usage:

```
Sampler$format()
```

Method `print()`: Printer.

Usage:

`Sampler$print(...)`

Arguments:

... (ignored).

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

Usage:

`Sampler$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#)

Sampler1D

Sampler1D Class

Description

1D sampler, abstract base class for Sampler like [Sampler1DUnif](#), [Sampler1DRfun](#), [Sampler1DCateg](#) and [Sampler1DNormal](#).

Super class

[paradox::Sampler](#) -> Sampler1D

Active bindings

`param` ([Param](#))

Returns the one Parameter that is sampled from.

Methods

Public methods:

- [Sampler1D\\$new\(\)](#)
- [Sampler1D\\$clone\(\)](#)

Method `new()`: Creates a new instance of this R6 class.

Note that this object is typically constructed via derived classes, e.g., [Sampler1DUnif](#).

Usage:

`Sampler1D$new(param)`

Arguments:

param ([Param](#))
 Domain / support of the distribution we want to sample from.

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

Usage:
 Sampler1D\$clone(deep = FALSE)

Arguments:
 deep Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

Sampler1DCateg	<i>Sampler1DCateg Class</i>
----------------	-----------------------------

Description

Sampling from a discrete distribution, for a [ParamFct](#) or [ParamLgl](#).

Super classes

[paradox::Sampler](#) -> [paradox::Sampler1D](#) -> Sampler1DCateg

Public fields

prob (numeric() | NULL)
 Numeric vector of param\$nlevels probabilities.

Methods

Public methods:

- [Sampler1DCateg\\$new\(\)](#)
- [Sampler1DCateg\\$clone\(\)](#)

Method new(): Creates a new instance of this [R6](#) class.

Usage:
 Sampler1DCateg\$new(param, prob = NULL)

Arguments:
 param ([Param](#))
 Domain / support of the distribution we want to sample from.
 prob (numeric() | NULL)
 Numeric vector of param\$nlevels probabilities, which is uniform by default.

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

Usage:

```
Sampler1DCateg$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

Sampler1DNormal	<i>Sampler1DNormal Class</i>
-----------------	------------------------------

Description

Normal sampling (potentially truncated) for [ParamDbl](#).

Super classes

```
paradox::Sampler -> paradox::Sampler1D -> paradox::Sampler1DRfun -> Sampler1DNormal
```

Active bindings

```
mean (numeric(1))
  Mean parameter of the normal distribution.
sd (numeric(1))
  SD parameter of the normal distribution.
```

Methods

Public methods:

- [Sampler1DNormal\\$new\(\)](#)
- [Sampler1DNormal\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

```
Sampler1DNormal$new(param, mean = NULL, sd = NULL)
```

Arguments:

`param` ([Param](#))

Domain / support of the distribution we want to sample from.

`mean` (numeric(1))

Mean parameter of the normal distribution. Default is `mean(c(param$lower, param$upper))`.

`sd` (numeric(1))

SD parameter of the normal distribution. Default is `(param$upper - param$lower) / 4`.

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

Usage:

```
Sampler1DNormal$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

Sampler1DRfun

Sampler1DRfun Class

Description

Arbitrary sampling from 1D RNG functions from R.

Super classes

```
paradox::Sampler -> paradox::Sampler1D -> Sampler1DRfun
```

Public fields

`rfun` (function())

Random number generator function.

`trunc` (logical(1))

TRUE enables naive rejection sampling, so we stay inside of [lower, upper].

Methods

Public methods:

- [Sampler1DRfun\\$new\(\)](#)
- [Sampler1DRfun\\$clone\(\)](#)

Method `new()`: Creates a new instance of this R6 class.

Usage:

```
Sampler1DRfun$new(param, rfun, trunc = TRUE)
```

Arguments:

`param` ([Param](#))

Domain / support of the distribution we want to sample from.

`rfun` (function())

Random number generator function, e.g. `rexp` to sample from exponential distribution.

`trunc` (logical(1))

TRUE enables naive rejection sampling, so we stay inside of [lower, upper].

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

Usage:

`Sampler1DRfun$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

Sampler1DUnif

Sampler1DUnif Class

Description

Uniform random sampler for arbitrary (bounded) parameters.

Super classes

[paradox::Sampler](#) -> [paradox::Sampler1D](#) -> [Sampler1DUnif](#)

Methods

Public methods:

- [Sampler1DUnif\\$new\(\)](#)
- [Sampler1DUnif\\$clone\(\)](#)

Method `new()`: Creates a new instance of this R6 class.

Usage:

`Sampler1DUnif$new(param)`

Arguments:

`param` ([Param](#))

Domain / support of the distribution we want to sample from.

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

Usage:

`Sampler1DUnif$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

SamplerHierarchical *SamplerHierarchical Class*

Description

Hierarchical sampling for arbitrary param sets with dependencies, where the user specifies 1D samplers per param. Dependencies are topologically sorted, parameters are then sampled in topological order, and if dependencies do not hold, values are set to NA in the resulting data . table.

Super class

`paradox::Sampler` -> SamplerHierarchical

Public fields

`samplers (list())`
List of [Sampler1D](#) objects that gives a Sampler for each [Param](#) in the `param_set`.

Methods

Public methods:

- `SamplerHierarchical$new()`
- `SamplerHierarchical$clone()`

Method `new()`: Creates a new instance of this R6 class.

Usage:

`SamplerHierarchical$new(param_set, samplers)`

Arguments:

`param_set` ([ParamSet](#))

Domain / support of the distribution we want to sample from. ParamSet is cloned on construction.

`samplers (list())`

List of [Sampler1D](#) objects that gives a Sampler for each [Param](#) in the `param_set`.

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

Usage:

`SamplerHierarchical$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerJointIndep](#), [SamplerUnif](#), [Sampler](#)

SamplerJointIndep *SamplerJointIndep Class*

Description

Create joint, independent sampler out of multiple other samplers.

Super class

`paradox::Sampler` -> SamplerJointIndep

Public fields

samplers (list())
List of [Sampler](#) objects.

Methods

Public methods:

- [SamplerJointIndep\\$new\(\)](#)
- [SamplerJointIndep\\$clone\(\)](#)

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

```
SamplerJointIndep$new(samplers)
```

Arguments:

samplers (list())
List of [Sampler](#) objects.

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerUnif](#), [Sampler](#)

SamplerUnif

SamplerUnif Class

Description

Uniform random sampling for an arbitrary (bounded) [ParamSet](#). Constructs 1 uniform sampler per [Param](#), then passes them to [SamplerHierarchical](#). Hence, also works for [ParamSets](#) sets with dependencies.

Super classes

`paradox::Sampler -> paradox::SamplerHierarchical -> SamplerUnif`

Methods

Public methods:

- `SamplerUnif$new()`
- `SamplerUnif$clone()`

Method `new()`: Creates a new instance of this [R6](#) class.

Usage:

```
SamplerUnif$new(param_set)
```

Arguments:

`param_set` ([ParamSet](#))

Domain / support of the distribution we want to sample from. [ParamSet](#) is cloned on construction.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Other Sampler: [Sampler1DCateg](#), [Sampler1DNormal](#), [Sampler1DRfun](#), [Sampler1DUnif](#), [Sampler1D](#), [SamplerHierarchical](#), [SamplerJointIndep](#), [Sampler](#)

transpose	<i>transpose</i>
-----------	------------------

Description

Converts [data.table::data.table](#) into a list of lists of points, possibly removes NA entries of inactive parameter values due to unsatisfied dependencies, and possibly calls the trafo function of the [ParamSet](#).

Usage

```
transpose(data, ps = NULL, filter_na = TRUE, trafo = TRUE)
```

Arguments

data	(data.table::data.table) Rows are points and columns are parameters.
ps	(ParamSet) If trafo = TRUE, used to call trafo function.
filter_na	(logical(1)) Should NA entries of inactive parameter values be removed due to unsatisfied dependencies?
trafo	(logical(1)) Should the trafo function of the ParamSet be called?

Index

- * **Params**
 - Param, 10
 - ParamDbl, 13
 - ParamFct, 15
 - ParamInt, 17
 - ParamLgl, 19
 - ParamUty, 28
- * **Sampler**
 - Sampler, 30
 - Sampler1D, 31
 - Sampler1DCateg, 32
 - Sampler1DNormal, 33
 - Sampler1DRfun, 34
 - Sampler1DUnif, 35
 - SamplerHierarchical, 36
 - SamplerJointIndep, 37
 - SamplerUnif, 38
- * **generate_design**
 - generate_design_grid, 7
 - generate_design_lhs, 8
 - generate_design_random, 9
- assert_param, 3
- assert_param_set (assert_param), 3
- CondAnyOf (Condition), 4
- CondEqual (Condition), 4
- Condition, 4, 21, 25, 27
- data.table::data.table, 24, 25, 39
- data.table::data.table(), 5, 6, 10, 20, 21, 27
- Design, 5, 7–9, 30
- generate_design_grid, 7, 8, 9
- generate_design_grid(), 5
- generate_design_lhs, 7, 8, 9
- generate_design_random, 7, 8, 9
- is_noddefault (NO_DEF), 9
- lhs::maximinLHS(), 8
- NO_DEF, 9, 11, 14, 16, 18, 20, 29
- NoDefault (NO_DEF), 9
- paradox (paradox-package), 2
- paradox-package, 2
- paradox::Param, 13, 15, 17, 19, 28
- paradox::ParamSet, 26
- paradox::Sampler, 31–38
- paradox::Sampler1D, 32–35
- paradox::Sampler1DRfun, 33
- paradox::SamplerHierarchical, 38
- Param, 3, 7, 9, 10, 10, 13–21, 23, 27–29, 31–36, 38
- ParamDbl, 10, 11, 13, 13, 15–20, 22, 28, 29, 33
- ParamFct, 10, 13–15, 15, 17–20, 22, 28, 29, 32
- ParamInt, 13–17, 17, 19, 20, 22, 28, 29
- ParamLgl, 13–19, 19, 22, 28, 29, 32
- ParamSet, 3, 5–12, 14, 16, 18, 20, 20, 22, 23, 26, 27, 29, 30, 36, 38, 39
- ParamSetCollection, 26, 27
- ParamUty, 13–20, 28, 28
- R6, 4, 6, 11, 13, 15, 17, 19, 23, 27, 29–38
- Sampler, 30, 32–38
- Sampler1D, 30, 31, 31, 33–38
- Sampler1DCateg, 31, 32, 32, 34–38
- Sampler1DNormal, 31–33, 33, 35–38
- Sampler1DRfun, 31–34, 34, 35–38
- Sampler1DUnif, 31–35, 35, 36–38
- SamplerHierarchical, 30–35, 36, 37, 38
- SamplerJointIndep, 30–36, 37, 38
- SamplerUnif, 9, 31–37, 38
- transpose, 39
- vname, 24, 25