Package 'paradox'

July 12, 2019

Type Package

Title Define and Work with Parameter Spaces for Complex Algorithms

Version 0.1.0

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

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

Imports backports, checkmate, data.table, mlr3misc, R6

Suggests lhs, testthat

Encoding UTF-8

NeedsCompilation no

RoxygenNote 6.1.1

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' '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]

Maintainer Michel Lang <michellang@gmail.com>

Repository CRAN

Date/Publication 2019-07-12 14:50:02 UTC

R topics documented:

`paradox-package` *paradox: Define and Work with Parameter Spaces for Complex Algorithms*

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> (0000-0001-9754-0393)

Authors:

- Bernd Bischl <bernd_bischl@gmx.net> (0000-0001-6002-6980)
- Jakob Richter <jakob1richter@gmail.com> (0000-0003-4481-5554)
- Xudong Sun <smilesun.east@gmail.com> (0000-0003-3269-2307)
- Martin Binder <mlr.developer@mb706.com>

See Also

Useful links:

- https://paradox.mlr-org.com
- 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`	(`character()`) Allowed subclasses.
`no_untyped`	(`logical(1)`) Are untyped Params allowed?
`must_bounded`	(`logical(1)`) Only bounded Params allowed?
`param_set`	ParamSet.
`no_deps`	(`logical(1)`) Are dependencies allowed?

Value

The checked object, invisibly.

Condition *Dependency Condition*

Description

Condition object, to specify the condition in a dependency.

Format

R6::R6Class object.

Construction

```
c = Condition$new(type, rhs)
```

- `type` :: `character(1)`
  Name / type of the condition.
- `rhs::any`
  Right-hand-side of the condition.

Methods

- `test(function(x))`
  `??? -> logical(n)`
  Checks if condition is satisfied. Called on a vector of parent param values.

Currently implemented simple conditions

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

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.

Format

R6::R6Class object.

Construction

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

Note that the first 2 arguments are NOT cloned during construction!

- `param_set` :: `ParamSet`.
- `data` :: `data.table::data.table()`
  Right hand side of the condition.
- `remove_dupl` :: `logical(1)`
  Remove duplicates?

Fields

- `param_set` :: `ParamSet`.
- `data` :: `data.table::data.table()`
  Stored `data`.

Methods

- `transpose(filter_na = TRUE, trafo = TRUE)`
  (`logical(1)`, `logical(1)`) -> `list()` of `list()`
  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.

`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

(data.table::data.table()).

See Also

Other generate_design: generate_design_lhs, generate_design_random

Examples

```
ps = ParamSet$new(list(
  ParamDbl$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

(data.table::data.table()).

See Also

Other generate_design: generate_design_grid, generate_design_random

Examples

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

`param_set`	:: ParamSet.
`n`	:: `integer(1)` Number of points to draw randomly.

Value

(`data.table::data.table()`).

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_nodefault()`: Is an object of type 'no default'?

Param *Param Object*

Description

Abstract base class for parameters.

Format

R6::R6Class object.

Construction

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

- `id` :: `character(1)`
  ID of this parameter.
- `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.
- `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.

Fields

- `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?
- `storage_type` :: `character(1)`
  Data type when values of this parameter are stored in a data table or sampled.

Methods

- `test(x)`, `check(x)`, `assert(x)`
  Three **checkmate**-like check-functions. 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`.
- `qunif(x)`
  `numeric(n)` -> `vector(n)`
  Takes values from [0,1] and map 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.
- `rep(n)`
  `integer(1)` -> ParamSet
  Repeats this parameter n-times (by cloning). Each parameter is named "<id>*rep*<k>" and gets the additional tag "<id>_rep".

S3 methods

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

See Also

Other Params: `ParamDbl`, `ParamFct`, `ParamInt`, `ParamLgl`, `ParamUty`

`ParamDbl` *Numerical Parameter*

Description

A Param to describe real-valued parameters.

Format

R6::R6Class object inheriting from Param.

Construction

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

Arguments of Param, and additionally:

- `lower` :: `numeric(1)`
  Lower bound, can be `-Inf`.
- `upper` :: `numeric(1)`
  Upper bound can be `+Inf`.

Fields

Fields of Param, and additionally:

- `lower` :: `numeric(1)`
  Lower bound.
- `upper` :: `numeric(1)`
  Upper bound.
- `levels` :: `NULL`
  Allowed levels. Always `NULL` for this parameter.
- `nlevels` :: `Inf`
  Number of categorical levels. Always `Inf` for this parameter.
- `is_bounded` :: `logical(1)`
  Are the bounds finite?

Methods

See Param.

See Also

Other Params: `ParamFct`, `ParamInt`, `ParamLgl`, `ParamUty`, `Param`

Examples

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

ParamFct *Factor Parameter*

Description

A Param to describe categorical (factor) parameters.

Format

R6::R6Class object inheriting from Param.

Construction

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

Arguments of Param, and additionally:

- `levels` :: `character()`
  Set of allowed levels.

Fields

Fields of Param, and additionally:

- `lower` :: `numeric(1)`
  Lower bound. Always `NA` for this parameter.
- `upper` :: `numeric(1)`
  Upper bound. Always `NA` for this parameter.
- `levels` :: `character()`
  Allowed levels.
- `nlevels` :: `Inf`
  Number of categorical levels.
- `is_bounded` :: `TRUE`
  Are the bounds finite? Always `TRUE` for this parameter.

Methods

See Param.

See Also

Other Params: `ParamDbl`, `ParamInt`, `ParamLgl`, `ParamUty`, `Param`

Examples

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

ParamInt *Integer Parameter*

Description

A Param to describe integer parameters.

Format

R6::R6Class object inheriting from Param.

Construction

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

Arguments of Param, and additionally:

- `lower` :: `numeric(1)`
  Lower bound, can be `-Inf`.
- `upper` :: `numeric(1)`
  Upper bound can be `+Inf`.

Fields

Fields of Param, and additionally:

- `lower` :: `numeric(1)`
  Lower bound.
- `upper` :: `numeric(1)`
  Upper bound.
- `levels` :: `NULL`
  Allowed levels. Always `NULL` for this parameter.
- `nlevels` :: `integer(1)`
  Number of categorical levels. Here, the number integers in the range `[lower, upper]`, or `Inf` if unbounded.
- `is_bounded` :: `logical(1)`
  Are the bounds finite?

Methods

See Param.

See Also

Other Params: `ParamDbl`, `ParamFct`, `ParamLgl`, `ParamUty`, `Param`

Examples

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

ParamLgl *Logical Parameter*

Description

A Param to describe logical parameters.

Format

R6::R6Class object inheriting from Param.

Construction

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

See Arguments of Param.

Fields

Fields of Param, and additionally:

- `lower` :: `numeric(1)`
  Lower bound. Always `NA` for this parameter.
- `upper` :: `numeric(1)`
  Upper bound. Always `NA` for this parameter.
- `levels` :: `logical(2)`
  Allowed levels. Always `c(TRUE, FALSE)` for this parameter.
- `nlevels` :: `Inf`
  Number of categorical levels. Always 2 for this parameter.
- `is_bounded` :: `TRUE`
  Are the bounds finite? Always `TRUE` for this parameter.

Methods

See Param.

See Also

Other Params: `ParamDbl`, `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).

Usage

```
ParamSet
```

Format

An object of class `R6ClassGenerator` of length 24.

Construction

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

- `params` :: named `list()`
  List of Param, named with their respective ID. Parameters are cloned.

Fields

- `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(1)`
  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.
- `storage_type` :: `character()`
  Data types of parameters when stored in tables. Named with parameter IDs.
- `tags` :: named `list()` of `character()`
  Can be used to group and subset 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.
- is_number :: named `logical()`
  Position is TRUE for ParamDbl and ParamInt. Named with parameter IDs.
- is_categ :: named `logical`
  Position is TRUE for ParamFct and ParamLgl. Named with parameter IDs.
- `has_deps` :: `logical(1)`
  Has the set parameter dependencies?
- `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.
- `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?

Public methods

- `ids(class = NULL, is_bounded = NULL, tags = NULL)`
  (`character`, `logical(1)`, `character()`) -> `character()`
  Retrieves IDs of contained parameters based on some filter criteria selections, `NULL` means no restriction.
- `get_values(class = NULL, is_bounded = NULL, tags = NULL)`
  (`character()`, `logical(1)`, `character()`) -> named `list()`

Retrieves parameter values based on some selections, NULL means no restriction and is equivalent to $values.

- add(param_set)
  (Param | ParamSet) -> self
  Adds a single param or another set to this set, all params are cloned.
- subset(ids)
  character() -> self
  Changes the current set to the set of passed IDs.
- test(x), check(x), assert(x)
  Three **checkmate**-like check-functions. 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.
- add_dep(id, on, cond)
  (character(1), character(1), Condition) -> self
  Adds a dependency to this set, so that param id now depends on param on.

S3 methods and type converters

- as.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

Examples

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

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

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

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

`ParamSetCollection` *ParamSetCollection*

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.

Format

R6::R6Class object inheriting from ParamSet.

Construction

```
ParamSetCollection$new(sets)
```

- `sets` :: list of ParamSet
  Parameter objects are cloned.

Methods

- remove_sets(ids)
  `character()` -> `self`
  Removes sets of given ids from collection.

ParamUty *Untyped Parameter*

Description

A Param to describe untyped parameters.

Format

R6::R6Class object inheriting from Param.

Construction

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

Arguments of Param, and additionally:

- `custom_check` :: `function()`
  Custom function to check the feasibility. Defaults to `NULL`.

Fields

Fields of Param, and additionally:

- `lower` :: `numeric(1)`
  Lower bound. Always `NA` for this parameter.
- `upper` :: `numeric(1)`
  Upper bound. Always `NA` for this parameter.
- `levels` :: `NULL`
  Allowed levels. Always `NULL` for this parameter.
- `nlevels` :: `numeric(1)`
  Number of categorical levels. Always `Inf` for this parameter.
- `is_bounded` :: `FALSE`
  Are the bounds finite? Always `FALSE` for this parameter.

Methods

See Param.

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.

Format

R6::R6Class object.

Construction

Note: This object is typically constructed via a derived classes.

```
smpl = Sampler$new(param_set)
```

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

Fields

- `param_set` :: ParamSet
  Domain / support of the distribution we want to sample from.

Methods

- `sample(n)`
  `integer(1)` -> Design
  Sample `n` values from the distribution.

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.

Format

R6::R6Class inheriting from Sampler.

Construction

Note: This object is typically constructed via a derived classes, e.g. Sampler1DUnif, Sampler1DRfun, Sampler1DCateg or Sampler1DNormal.

```
smpl = Sampler1D$new(param)
```

- `param` :: Param
  Domain / support of the distribution we want to sample from.

Fields

See Sampler. Additionally, the class provides:

- `param` :: Param
  Returns the one Parameter that is sampled from.

Methods

See Sampler.

See Also

Other Sampler: `Sampler1DCateg`, `Sampler1DNormal`, `Sampler1DRfun`, `Sampler1DUnif`, `SamplerHierarchical`, `SamplerJointIndep`, `SamplerUnif`, `Sampler`

Sampler1DCateg *Sampler1DCateg Class*

Description

Sampling from a discrete distribution, for a ParamFct or ParamLgl.

Format

R6::R6Class inheriting from Sampler1D.

Construction

```
smpl = Sampler1DCateg$new(param, prob = NULL)
```

- `param` :: Param
  Domain / support of the distribution we want to sample from.
- `prob` :: `numeric()`
  Numeric vector of `param$nlevels` probabilities, which is uniform by default.

Fields

See Sampler1D. Additionally, the class provides:

- `prob` :: `numeric(n)`
  Numeric vector of `param$nlevels` probabilities, which is uniform by default.

Methods

See Sampler1D.

See Also

Other Sampler: `Sampler1DNormal`, `Sampler1DRfun`, `Sampler1DUnif`, `Sampler1D`, `SamplerHierarchical`, `SamplerJointIndep`, `SamplerUnif`, `Sampler`

`Sampler1DNormal` *Sampler1DNormal Class*

Description

Normal sampling (potentially truncated) for ParamDbl.

Format

R6::R6Class inheriting from Sampler1D.

Construction

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

- `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`.

Fields

See Sampler1D. Additionally, the class provides:

- `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`.

Methods

See Sampler1D.

See Also

Other Sampler: `Sampler1DCateg`, `Sampler1DRfun`, `Sampler1DUnif`, `Sampler1D`, `SamplerHierarchical`, `SamplerJointIndep`, `SamplerUnif`, `Sampler`

Sampler1DRfun *Sampler1DRfun Class*

Description

Arbitrary sampling from 1D RNG functions from R.

Format

R6::R6Class inheriting from Sampler1D.

Construction

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

- `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].

Fields

See Sampler1D. Additionally, the class provides:

- `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].

Methods

See Sampler1D.

See Also

Other Sampler: `Sampler1DCateg`, `Sampler1DNormal`, `Sampler1DUnif`, `Sampler1D`, `SamplerHierarchical`, `SamplerJointIndep`, `SamplerUnif`, `Sampler`

Sampler1DUnif *Sampler1DUnif Class*

Description

Uniform random sampler for arbitrary (bounded) parameters.

Format

R6::R6Class inheriting from Sampler1D.

Construction

```
smpl = Sampler1DUnif$new(param)
```

- `param` :: Param
  Domain / support of the distribution we want to sample from.

Fields

See Sampler1D.

Methods

See Sampler1D.

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`.

Format

R6::R6Class inheriting from Sampler.

Construction

```
smpl = SamplerHierarchical$new(param_set, samplers)
```

- `param_set` :: ParamSet
  Domain / support of the distribution we want to sample from.
- `samplers` :: `list()`
  List of Sampler1D objects that gives a Sampler for each Param in the `param_set`.

Fields

See Sampler. Additionally, the class provides:

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

Methods

See Sampler.

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.

Format

R6::R6Class inheriting from Sampler.

Construction

```
smpl = SamplerJointIndep$new(samplers)
```

- `samplers` :: `list()`
  List of Sampler objects.

Fields

See Sampler. Additionally, the class provides:

- `samplers` :: `list()`
  List of Sampler objects.

Methods

See Sampler.

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.

Format

R6::R6Class inheriting from SamplerHierarchical.

Construction

See Sampler.

Fields

See Sampler.

Methods

See Sampler.

See Also

Other Sampler: Sampler1DCateg, Sampler1DNormal, Sampler1DRfun, Sampler1DUnif, Sampler1D, SamplerHierarchical, SamplerJointIndep, Sampler

Index