Package ‘powergrid’

September 30, 2025

Title Power Analysis Across a Grid of Assumptions

Version 0.5.0

Description Evaluate a function across a grid of parameters. The function may be evaluated once, or many times for simulation. Parallel computing is facilitated. Utilities aim at performing analyses of power and sample size, allowing for easy search of minimum n (or min/max of any other parameter) to achieve a desired minimal level of power (or maximum of any other objective). Plotting functions are included that present the dependency of n and power in relation to further assumptions.

License GPL-3

Encoding UTF-8

RoxygenNote 7.3.3

Suggests future.apply (>= 1.11.2), future (>= 1.33.2), knitr, rmarkdown, testthat (>= 3.0.0)

Imports stats, methods

VignetteBuilder knitr

Config/testthat/edition 3

URL https://github.com/SwissClinicalTrialOrganisation/powergrid

BugReports https://github.com/SwissClinicalTrialOrganisation/powergrid/issues

NeedsCompilation no

Author Gilles Dutilh [aut, cre] (ORCID: <https://orcid.org/0000-0002-6719-2508>),
Richard Charles Allen [aut] (ORCID: <https://orcid.org/0000-0001-6012-7888>)

Maintainer Gilles Dutilh <info@gillesdutilh.com>

Repository CRAN

Date/Publication 2025-09-30 08:20:02 UTC

Contents

`AddExample` *Add an example to an existing PowerPlot or GridPlot*

Description

Add example arrow(s) to an existing figure created by PowerPlot or GridPlot.

`AddExample` is a higher level plotting function, so it does not know anything about the figure it draws on top off. Therefore, take care your figure makes sense, by supplying the same arguments `x` and `slicer` that you supplied to the `PowerPlot` or `link{GridPlot}` you are drawing on top off: With `slicer` you define the plotted plain, with `example` the value on the x-axis where the arrow starts. To be sure of a sensible result, use the argument `example` inside `Powerplot` or `GridPlot`.

Usage

```
AddExample(
  x,
  slicer = NULL,
  example = NULL,
  find_lowest = TRUE,
  target_value = NULL,
  target_at_least = TRUE,
  method = "step",
  summary_function = mean,
  col = grDevices::grey.colors(1, 0.2, 0.2),
  example_text = TRUE,
  ...
)
```

Arguments

`x, target_value, target_at_least, find_lowest, method, example_text, summary_function`
: See help for PowerPlot.

`slicer`
: A list, internally passed on to ArraySlicer to cut out a (multidimensional) slice from x. You can achieve the same by appending "slicing" inside argument `example`. However, to assure that the result of AddExample is consistent with the figure it draws on top of (PowerPlot or GridPlot), copy the arguments `x` and `slicer` given to PowerPlot or GridPlot to AddTarget.

`example`
: A list, defining at which value (list element value) of which parameter(s) (list element name(s)) the example is drawn for a power of `target_value`. You may supply par vector(s) longer than 1 for multiple examples. If `example` contains multiple parameters to define the example, all must contain a vector of the same length. Be aware that the first element of `example` defines the parameter x-axis, so this function is not fool proof. See argument `slicer` above. If x has only one dimention, the example needs not be defined.

`col`
: Color of arrow and text drawn.

`...`
: Further arguments are passed to the two calls of function `graphics::arrows` drawing the nicked arrow.

Details

arguments `slicer` and `example`:

`slicer` takes the slice of x that is in the figure, `example` defines at which value of which parameter, the example is drawn. These arguments' use is the same as in PowerPlot and GridPlot. If you want to make sure that the result of AddExample is consistent with a figure previously created using PowerPlot or GridPlot, copy the argument `slicer` to such function to AddExample, and define your example in `example`.

Note however, that:

slicer = list(a = c(1, 2)) and example = list(b = c(3, 4))

has the same result as:

example = list(b = c(3, 4) and a = c(1, 2)) (not defining slicer)

Importantly, the the order of `example` matters here, where the first element defines the x-axis.

multiple examples:

Argument `example` may contain vectors with length longer than one to draw multiple examples.

Value

invisibly NULL

Author(s)

Gilles Dutilh

See Also

PowerPlot, GridPlot

Examples

```
## For more examples, see ?PowerPlot

## Set up a grid of n, delta and sd:
sse_pars = list(
  n = seq(from = 10, to = 60, by = 4),
  delta = seq(from = 0.5, to = 1.5, by = 0.1), # effect size
  sd = seq(.1, 1.1, .2)) # Standard deviation
## Define a power function using these parameters:
PowFun <- function(n, delta, sd){ # power for a t-test at alpha = .05
  ptt = power.t.test(n = n/2, delta = delta, sd = sd,
                     sig.level = 0.05)
  return(ptt$power)
}
## Evaluate PowFun across the grid defined by sse_pars:
power_array = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = NA)

## ======================
## PowerPlot
## ======================
PowerPlot(power_array,
          slicer = list(sd = .7),
          )
AddExample(power_array,
           slicer = list(sd = .7), # be sure to cut out the same plain as above
           example = list(delta = .9),
           target_value = .9,
           col = 'blue')
AddExample(power_array,
           slicer = list(sd = .7),
           example = list(delta = c(.7, 1)), # multiple examples
           target_value = .9,
           col = 'yellow')
## Careful, you can move the slicer argument to example:
AddExample(power_array,
           example = list(delta = 1.2, sd = .7), # delta (x-axis) first
           target_value = .9,
           col = 'green')
## Careful, because you can put the wrong value on x-axis!
AddExample(power_array,
           example = list(sd = .7, delta = 1.2), # sd first?!
           target_value = .9,
           col = 'red')

## ======================
## GridPlot
## ======================
GridPlot(power_array, target_value = .9)
AddExample(power_array,
           example = list(delta = 1, sd = .7),
           target_value = .9
           )
```

```
## two examples
AddExample(power_array,
           example = list(delta = c(.9, 1.2), sd = c(.5, 1.1)),
           target_value = .9, col = 3
           )
```

`ArraySlicer` *Cut slice from array (typically of class* `power_array`*)*

Description

Cut out a slice from an array. The resulting slice may be single- or multidimensional. The function is intended for arrays of class "power_array", and makes sure that the resulting array is of class power_array and keeps and, where needed, updates the object's attributes. These attributes are needed for various functions in the powergrid package to work well.

Usage

```
ArraySlicer(x, slicer = NULL)
```

Arguments

`x`	An array, in most common use cases an array of class `power_array`, but may be any array with named dimensions.
`slicer`	A list whose named elements define at which dimension (the list element names), at which values (the list element values) a slice is taken from `power_array`. Default NULL returns the unchanged array.

Details

Internally, indexing ([) is used, but the implementation in ArraySlicer is very flexible allowing for any number of dimensions in any order in the `slicer` argument. The resulting slice is always an array, also if only one dimension is left. `dimnames` are kept intact.

Value

An array with reduced dimensions as given by `slicer`. Note that, relative to a standard array, some additional attributes are passed to be used in the functions in package `powergrid`

Author(s)

Gilles Dutilh

See Also

`PowerGrid`, `[.power_array` for reducing the dimensions of an array of class `power_array` using [-indexing.

Examples

```
sse_pars = list(
  n = seq(from = 20, to = 60, by = 5),
  delta = seq(from = 0.5, to = 1.5, by = 0.2),
  sd = seq(.1, .9, .2),
  alpha = c(.05, .025, .1)) # a 4-dimensional grid
PowFun <- function(n, delta, sd, alpha){
  ptt = power.t.test(n = n/2, delta = delta, sd = sd,
                     sig.level = alpha)
  return(ptt$power)
}
power_array = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = NA)
## cut out a 2-dimensional plane:
ArraySlicer(power_array,
            slicer = list(alpha = .1, sd = .9))
## Note that above, the dimension levels are called as numeric values, so the
## following works as well:
ArraySlicer(power_array,
            slicer = list(alpha = 0.1, sd = 0.9))
## They can be called by their actual character values as well:
ArraySlicer(power_array,
            slicer = list(alpha = '0.1', sd = '0.9'))
## (compare with dimnames(power_array))
## the following does not work:
## Not run:
ArraySlicer(power_array,
            slicer = list(alpha = '.1', sd = '.9'))

## End(Not run)
##
## Cut out multiple levels from one dimension
ArraySlicer(power_array,
            slicer = list(alpha = .1, sd = c(.9, .7)))
```

Example *Find combination of parameters required for achieving a desired power (or other objective).*

Description

Find combination of parameters yielding desired power (or any other target value) in an object of class "power_array".

Usage

```
Example(
  x,
  example = NULL,
  target_value = NULL,
```

```
  target_at_least = TRUE,
  find_lowest = TRUE,
  method = "step",
  summary_function = mean
)
```

Arguments

`x` Object of class `power_array`

`example` List with named elements representing the constellation of parameter values for which the example should be found. The names of this list should match the dimension names of `x`, their values should be exact values available at these dimensions. See example for an illustration.

`target_value` Which value (of typically power) should be achieved at the example.

`target_at_least` Logical. Set to TRUE if you aim to achieve a minimum value (e.g., a power must be *at least* 90%), or FALSE if you want to allow a maximum value (e.g., the width of the expected CI may be *at most* a certain value).

`find_lowest` Logical, indicating whether the example should be found that minimizes a parameter (typically: minimal required n) to achieve the `target_value` or maximizes this assumption (e.g., maximal allowed SD).

`method` Character string, indicating how the location of the example is found, passed on internally to `FindTarget`. Either "step": walking in steps along the parameter of interest or "lm": Interpolating assuming a linear relation between the parameter of interest and (qnorm(x) + qnorm(1

- 0.05)) ^ 2. This method "lm" is inspired on the implementation in the sse package by Thomas Fabbro.

`summary_function` When x' attribute `summarized` is FALSE, `x` is summarized across iterations using this function before searching the example.

Details

In the most typical use case, and this is also the default, `Example` searches the *minimal* n where the power is *at least* equal to the value given by argument `target`. The function is, however, designed much more generically. The explanation below may be less helpful than trying the examples, but for completeness:

Argument `example` slices out a vector from object `x`, representing the values at the parameter combination given in example, thus, along the remaining parameter. Then, `Example` searches along this vector for the *minimal* parameter value where the value of the vector is *at least* equal to `target`. Thus, if the sliced out vector contains values of "power" along the parameter "effect size", it searches the minimal effect size at which the target power is achieved.

Two complications are made to allow for complete flexibility:

1. In the above description, *minimal* can be changed to *maximal* by setting argument `find_lowest` to FALSE. This is useful in the situation where one, e.g., searches for the highest standard deviation at which it is still possible to find a desirable power.

2. In the above description, *at least* can be changed to *at most* by setting target_at_least to FALSE. This allows to search, for example, for the minimal sample size where the expected confidence interval is smaller than a certain desired width.

Example searches for the minimum or maximum on one parameters (say, the minimum n) given *one single constellation* of further parameters. However, you may want to study how, say, the required n (or any other value) depends on the value of further parameters. The functions PowerPlot and GridPlot offer plotting functionalities to graphically illustrate such dependencies. If you want to find "Examples" as a function of parameter settings and work with these, you can use the workhorse behind 'Example', PowerPlot and Gridplot, FindTarget

Value

Example returns a list containing:

- "requested_example": the parameter combination at which the power (or whatever the values represent) was searched to achieve level target_value (typically the minimal power, e.g., .9), searching along parameter required name (typically n).
- "objective": was required_name searched to find the "min" or "max" of x?
- "target_value": which value should the power (or any other value) have?
- "required_name": the parameter searched along to find the minimum (or maximized if slot searched = 'max') to achieve objective. (typically n)
- "required_value": the minimum (or maximum if searched = "max") for parameter required_name (which is typically n)
- "searched": was the "min" or "max" for required_name searched?
- "target_at_least": Is the target_value a minimum (TRUE, as typical for power) or a maximum (FALSE, e.g., an expected uncertainty level)?

Author(s)

Gilles Dutilh

See Also

PowerGrid, FindTarget, PowerPlot, GridPlot

Examples

```
## ============================================
## Typical use case: find lowest n for a certain target power
## ============================================
sse_pars = list(
  n = seq(from = 10, to = 60, by = 2),
  delta = seq(from = 0.5, to = 1.5, by = 0.1), ## effect size
  sd = seq(.1, .9, .2)) ## Standard deviation
PowFun <- function(n, delta, sd){
  ptt = power.t.test(n = n/2, delta = delta, sd = sd,
                     sig.level = 0.05)
  return(ptt$power)
```

```
}
power_array = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = NA)
##'
ex_out = Example(power_array,
                 example = list(delta = .7, sd = .7),
                 target_value = .9)
ex_out #

## ============================================
## Illustration argument `find_lowest`
## ============================================
##
## In this example, we search for the *highest sd* for which the power is at
## least .9.
ex_out = Example(power_array,
                 example = list(n = 40, delta = .7),
                 target_value = .9, find_lowest = FALSE)
ex_out # note how the printed result indicates it searched for a maximal
                                        # permissible sd.

## ============================================
## Illustration argument `target_at_least`
## ============================================
##
## In the example below, we search for the lowest n where the expected CI-width
## is not larger than .88.
PowFun <- function(n, delta, sd){
  x1 = rnorm(n = n/2, sd = sd)
  x2 = rnorm(n = n/2, mean = delta, sd = sd)
  CI_width = diff(t.test(x1, x2)$conf.int) # CI95 is saved
}
sse_pars = list(
  n = seq(from = 10, to = 60, by = 5),
  delta = seq(from = 0.5, to = 1.5, by = 0.2),
  sd = seq(.5, 1.5, .2))
## we iterate, and take the average across iterations to get expected CI-width:
n_iter = 20
set.seed(1)
power_array = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = n_iter)
summary(power_array)
## Now, find lowest n for which the average CI width is *smaller than .88*.
ex_out = Example(power_array,
                 example = list(delta = .7, sd = .7),
                 target_value = .88,
                 find_lowest = TRUE, # we search the *lowest* n
                 target_at_least = FALSE # for a *maximal* mean CI width
                 )
ex_out # note how the printed result indicates the target CI is a maximum.

## ============================================
## When both `find_lowest` and `target_at_least` are FALSE
## ============================================
##
```

```
## In this example, we search for the *highest sd* for which the average CI
## width is still *smaller than or equal to .88*.
ex_out = Example(power_array,
                 example = list(delta = .7, n = 60),
                 target_value = .88,
                 find_lowest = FALSE, # we search the *highest* sd
                 target_at_least = FALSE # for a *maximal* mean CI width
                 )

ex_out # note how the printed result indicates that the *maximal permissible SD*
       # was found for a CI of *at most .88*.
```

FindTarget *Find requirements for target power (or other objective)*

Description

For most use cases of powergrid, you will not need this function, but rather use more convenient functions, most notable `Example`. `Example` shows you the smallest sample size to still find enough power, or the largest standard deviation at which your CI95 does not get too large. More insight about the relation between parameters and the resulting power may be gained with `PowerPlot` or `GridPlot`.

Only if you need to work with, say, the required n for a range of assumptions over and above PowerPlot and GridPlot, you will need to use FindTarget.

FindTarget takes as input an array (typically of class `power_array`). FindTarget then searches (up or down) along one chosen dimension for a value that meets a set target value (at least or at most). It does so for each combination of the remaining dimensions. Concretely, this may mean: The array contains the calculated power for each combination of dimensions n, effect size, and SD. The function may then find, for each combination of effect size and SD, the lowest n for which power of at least, say, .8 is achieved. The result would be an array of effect size by SD, containing the n's yielding acceptable power.

Usage

```
FindTarget(
  x,
  par_to_search = "n",
  find_lowest = TRUE,
  target_value = 0.9,
  target_at_least = TRUE,
  method = "step"
)
```

Arguments

- `x` — An array, most commonly of class `power_array`, possibly the result of taking a slice of an object of class `power_array` using `ArraySlicer` or the power_array []-indexing method.
- `par_to_search` — Which parameter should be searched to achieve the required target value. In the typical power analysis case, this is n.
- `find_lowest` — If TRUE, the lowest value of par_to_search is found that yields a value that meets the target. This is typical for n in a sample size estimation, where one searches the lowest n to achieve a certain power. For, e.g. the variance, one would however search for the maximum where the target power can still be achieved.
- `target_value` — The required value in x (e.g., .9, if the values represent power)
- `target_at_least` — Is the `target_value` a minimum (e.g., the power) or a maximum (e.g., the size of a confidence interval)
- `method` — How is the required `par_to_search` to achieve `target_value` found. Either `'step'`: walking in steps along `par_to_search` or `'lm'`: Interpolating assuming a linear relation between `par_to_search` and `(qnorm(x) + qnorm(1 - 0.05)) ^ 2`. Setting 'lm' is inspired on the implementation in the sse package by Thomas Fabbro.

Details

By default FindTarget searches along the dimension called `n` (`par_to_search`), searching for the lowest value (`find_lowest` = TRUE) where the array contains a value of at least (`target_at_least` = TRUE) .9 (the `target_value`), thus finding the minimal sample size required to achieve a power of 90%. These arguments may seem a bit confusing at first, but they allow for three additional purposes:

First, the implementation also allows to search for a value that is *at most* the `target_value`, by setting `target_at_least` to FALSE. This may be used, for example, when the aim is to find a sample size yielding a confidence interval that is not bigger than some maximum width.

Second, the implementation allows to search along another *named* dimension of x than n.

Third, the implementation allows to search for a certain target value to be achieved by maximizing (find_lowest = FALSE) the parameter on the searched dimension. This may be used, for example, when the aim is to find the maximum standard deviation at which a study's power is still acceptable.

`FindTarget` is most often called as the workhorse of `Example`, `PowerPlot` or `GridPlot`.

Value

Returns an array or vector: containing the value that is found for the par_to_search (say, n) meeting the target following above criteria (say, the lowest n for which the power is larger than .9), for each crossing of the levels of the other dimensions (say, delta, SD).

Author(s)

Gilles Dutilh

See Also

PowerGrid, Example, PowerPlot

Examples

```
## ============================================
## A basic power analysis example:
## ============================================
sse_pars = list(
  n = seq(from = 10, to = 60, by = 2),
  sig_level = seq(.01, .1, .01),
  delta = seq(from = 0.5, to = 1.5, by = 0.2), ## effect size
  sd = seq(.1, .9, .2)) ## Standard deviation
PowFun <- function(n, sig_level, delta, sd){
  ptt = power.t.test(n = n/2, delta = delta, sd = sd,
                     sig.level = sig_level)
  return(ptt$power)
}
power_array = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = NA)
summary(power_array) # four dimensions

## We can use Example so find the required sample size, but only for one example:
Example(power_array,
        example = list(delta = .7, sd = .7, sig_level = .05),
        target_value = .9)

## If we want to see the required sample size for all delta's, we can use
## FindTarget. Get the minimal n needed for achieving a value of 0.9, at sd =
## .3:
n_by_delta_sd_03 = FindTarget(power_array[, sig_level = '0.05', , sd = '0.3'],
                              par_to_search = 'n',
                              target_value = .9)

n_by_delta_sd_03
## just as an illustration, a figure (that can be much more aestetically made
## using PowerPlot)
plot(as.numeric(names(n_by_delta_sd_03)),
     n_by_delta_sd_03, type = 'l')

## =================================
## Higher dimensionality
## =================================

## The function works also for higher dimensionality:
n_by_delta_sd = FindTarget(power_array,
                           par_to_search = 'n',
                           target_value = .85)
## what is the minimum n to achieve .85 for different values of delta, sd,
## when  sig_level = 0.05:
n_by_delta_sd[5, , ] # note that for some combinations of delta and sd, there is
                     # no n yielding the required power at this significance
                     # level (NAs).
```

GridPlot *Plot requirements for achieving a target power as a function of assumptions about two parameters*

Description

Plots how the required sample size (or any other parameter) to achieve a certain power (or other objective) depends on two furhter parameters.

Usage

```
GridPlot(
  x,
  slicer = NULL,
  y_par = NULL,
  x_par = NULL,
  l_par = NULL,
  example = NULL,
  find_lowest = TRUE,
  target_value = 0.9,
  target_at_least = TRUE,
  method = "step",
  summary_function = mean,
  col = NULL,
  example_text = TRUE,
  title = NULL,
  par_labels = NULL,
  xlim = NULL,
  ylim = NULL,
  smooth = FALSE
)
```

Arguments

x	An object of class "power_array" (from `powergrid`).
slicer	If the parameter grid of `x` has more than 3 dimensions, a 3-dimensional slice must be cut out using `slicer`, a list whose elements define at which values (the list element value) of which parameter (the list element name) the slice should be cut.
y_par	Which parameter is searched for the minimum (or maximum if find_lowest == FALSE) yielding the target value; and shown on the y-axis. If NULL, `y_par` is set to the first, `x_par` to the second, and `l_par` to the third dimension name of 3-dimensional array `x`. If you want another than the first dimension as `y_par`, you need to see `y_par`, `x_par`, and `l_par` explicitly.
x_par, l_par	Which parameter is varied on the x-axis, and between lines, respectively. If none of `y_par`, `x_par` and `l_par` are given, the first, second, and third dimension of x are mapped to y_par, x_par, and l_par, respectively.

example
: A list defining for which combination of levels of `l_par` and `x_par` an example arrow should be drawn. List element names indicate the parameter, element value indicate the values at which the example is drawn.

find_lowest
: Logical, indicating whether the example should be found that minimizes an assumption (e.g., minimal required n) to achieve the `target_value` or an example that maximizes this assumption (e.g., maximally allowed SD).

target_value
: The target power (or any other value stored in x) that should be matched.

target_at_least
: Logical. Should `target_value` be minimally achieved (e.g., power), or maximially allowed (e.g., estimation uncertainty).

method
: The method to find the required parameter values, see `Example` and `FindTarget`.

summary_function
: If `x` is an object of class `power_array` where attribute `summarized` is FALSE (indicating individual iterations are stored in dimension `iter`, the iterations dimension is aggregated by `summary_fun`. Otherwise ignored.

col
: A vector with the length of `l_par` defining the color(s) of the lines.

example_text
: When an example is drawn, should the the required par value, and the line parameter value be printed alongside the arrow(s)

title
: Character string, if not `NULL`, replaces default figure title.

par_labels
: Named vector where elements names represent the parameters that are plotted, and the values set the desired labels.

xlim, ylim
: See `?graphics::plot`.

smooth
: Logical. If TRUE, a 5th order polynomial is fitted though the points constituting each line for smoothing.

Details

In the most typical use case, the y-axis shows the *minimal* sample size required to achieve a power of *at least* `target_value`, assuming the value of a parameter on the x-axis, and the value of another parameter represented by each line.

The use of this function is, however, not limited to finding a minimum n to achieve at least a certain power. See help of `Example` to understand the use of `target_at_least` and `fin_min`.

If the input to argument x (class `power_array`) contains iterations that are not summarized, it will be summarized by `summary_function` with default `mean`.

Note that a line may stop in a corner of the plotting region, not reaching the margin. This is often correct behavior, when the `target_value` level is not reached anywhere in that corner of the parameter range. In case n is on the y-axis, this may easily be solved by adding larger sample sizes to the grid (consider `Update`), and then adjusting the y-limit to only include the values of interest.

Value

A list with graphical information to use in further plotting.

Author(s)

Gilles Dutilh

See Also

PowerGrid, AddExample, Example, PowerPlot for similar plotting of just 2 parameters, at multiple power (target value) levels.

Examples

```
sse_pars = list(
  n = seq(from = 2, to = 100, by = 2),
  delta = seq(from = 0.1, to = 1.5, by = 0.05), ## effect size
  sd = seq(.1, .9, .1)) ## Standard deviation
PowFun <- function(n, delta, sd){
  ptt = power.t.test(n = n/2, delta = delta, sd = sd,
                     sig.level = 0.05)
  return(ptt$power)
}
power_array = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = NA)
GridPlot(power_array, target_value = .8)
## If that's too many lines, cut out a desired number of slices
GridPlot(power_array,
         slicer = list(sd = seq(.1, .9, .2)),
         target_value = .8)

## adjust labels, add example
GridPlot(power_array, target_value = .9,
         slicer = list(sd = seq(.1, .9, .2)),
         y_par = 'n',
         x_par = 'delta',
         l_par = 'sd',
         par_labels = c('n' = 'Sample Size',
                        'delta' = 'Arm Difference',
                        'sd' = 'Standard Deviation'),
         example = list(sd = .7, delta = .6))
## add additional examples useing AddExample. Note that these do not contain
## info about the line they refer to.
AddExample(power_array,
         target_value = .9,
         example = list(delta = c(.5, .8), sd = c(.3, .7)),
         col = 3
         )


## Above, GridPlot used the default: The first dimension is what you search
## (often n), the 2nd and 3rd define the grid of parameters at which the
#search # is done. Setting this explicitly, with x, y, and l-par, it looks
#like:
GridPlot(power_array, target_value = .8,
         slicer = list(sd = seq(.1, .9, .2)),
         y_par = 'n', # search the smallest n where target value is achieved
         x_par = 'delta',
         l_par = 'sd')

## You may also want to have different parameters on lines and axes:
```

```
GridPlot(power_array, target_value = .8,
         y_par = 'delta', # search the smallest delta where target value is achieved
         x_par = 'sd',
         l_par = 'n')
## Too many lines! Take some slices again:
GridPlot(power_array, target_value = .9,
         slicer = list(n = c(seq(10, 70, 10))),
         y_par = 'delta',
         x_par = 'sd',
         l_par = 'n', method = 'step')
```

PowerDF *Transform power_array into power_df*

Description

Transforms an object of class `power_array` to a data.frame, where values are stored in column x, and all other dimensions are columns. Some may find this "more tidy" to work with.

The class of the data.frame becomes ‘c("power_df", "data.frame"), enabling generics for data.frame. Note that the class "power_df" has currently no use but is included for future compatibility.

Usage

```
PowerDF(x)
```

Arguments

x	Object of class `power_array`

Value

An object of with classes c("power_df", "data.frame"), with the same attributes as `x`, aside from array-native attributes (dimnames, dim), plus the data.frame attributes `names` and `row_names`.

Author(s)

Gilles Dutilh

See Also

PowerGrid

Examples

```
## Define grid of assumptions to study:
sse_pars = list(
  n = seq(from = 10, to = 50, by = 20),         # sample size
  delta = seq(from = 0.5, to = 1.5, by = 0.5), # effect size
  sd = seq(.1, 1, .3))                        # standard deviation

## Define function that calculates power based on these assumptions:
PowFun <- function(n, delta, sd){
  ptt = power.t.test(n = n/2, delta = delta, sd = sd,
                     sig.level = 0.05)
  return(ptt$power)
}

## Evaluate at each combination of assumptions:
powarr = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = NA)
print(PowerDF(powarr))
```

`PowerGrid` *Evaluate function (iteratively) at a grid of input arguments*

Description

PowerGrid is an apply-like function, allowing to evaluate a function at the crossings of a set of parameters. The result is saved in an array with attributes that optimize further usage by functions in package `powergrid`. In particular, performing a function iteratively (using parallel computing if required) is implemented conveniently. The typical use is for evaluating statistical power at a grid of assumed parameters.

Usage

```
PowerGrid(
  pars,
  fun,
  more_args = NULL,
  n_iter = NA,
  summarize = TRUE,
  summary_function = mean,
  parallel = FALSE,
  n_cores = future::availableCores() - 1
)
```

Arguments

`pars` A list where each element is a numeric vector of values named as one of the arguments of `fun`. `fun` is applied to the full grid crossing the values of each of these parameters. If you aim to study other than numeric parameters, see details.

fun	A function to be applied at each combination of pars. Arguments may contain all element names of pars and more_args. Output should always be a numeric vector, typically of length one. However, a if you want to work with multiple outpus, each can be an element of the returned numeric vector.
more_args	Fixed arguments to fun that are not in pars. (internally used in .mapply for supplying argument MoreArgs)
n_iter	If not NA, function fun is applied n_iter times at each point in the grid defined by pars.
summarize	Logical indicating whether iterations (if n_iter is given) are to be summarized by summary_function.
summary_function	A function to be applied to aggregate across iterations. Defaults to mean, ignored when keep_iters == TRUE or when is.na(n_iter).
parallel	Logical indicating whether parallel computing should be applied. If TRUE, future::future_replicate is used internally.
n_cores	Passed on to future_replicate

Details

Function fun is evaluated at each combination of the argument values listed in pars and its results are stored in an array of class power_array, whose dimensions (and dimnames()) are defined by pars. For this to work, the element names of pars must match the argument names of fun.

Further arguments to fun:

If input parameters to fun are not to be part of the grid, but rather further settings, these can be passed on to fun through the argument more_args as a list with names reflecting the arguments of fun to be set.

Storing multiple outputs from fun:

You may have a function fun that returns a vector with length larger than one, as long as it is a single vector. When fun returns a vector with length larger than one, the power_array will have an additional dimension fun_out, with levels named after the names of fun's return vector (if given).

Non-numeric parameters:

You may want to study the effect of non-numeric parameters. This option is not supported for the argument pars, since the essential powergrid functions link{Example}, link{PowerPlot}, and link{GridPlot} need a direction to search. Nonetheless, you can study non-numeric parameters by having function fun returning multiple values, as described above.

Evaluating a function over iterations:

If n_iter is not NA (the default) but an integer, function fun is evaluated n_iter times. This will add an additional dimension 'iter' to the resulting array of class power_array. If your simulation is heavy, you may wanna set parallel = TRUE and choose the n_cores, invoking parallel computing using tfuture::future_replicate.

You may summarize the object with individual iterations across these iterations using function SummarizeIterations. Note that both summarized and non-summarized output of PowerGrid have class power_array. The summary status is saved in the attributes. This allows the powergrid utilities Example, PowerPlot, and GridPlot to do something sensible also with non-summarized objects.

Reproducibility:

The current status of .Random.seed is stored in the attribute random_seed (which is a list). To reproduce a call of PowerGrid involving randomness, precede new call to PowerGrid by .Random.seed = attr(<your_power_array>,which = 'random.seed')[[1]]. Note that if you Refine() your power_array, the .Random.seed at the moment of updating is appended to the random.seed attribute. So, to reconstruct a refined power_array, run the original call to PowerGrid after .Random.seed = attr(<your_power_array>, which = 'random.seed')[[1]], and the the call to Refine after .Random.seed = attr(<your_power_array>, which = 'random.seed')[[2]], etc.

Value

An array of class "power_array", with attributes containing informations about input arguments, summary status, the presence of multiple function outputs and more. This object class is handled sensibly by functions in package powergrid, including Example, PowerPlot, and GridPlot.

Author(s)

Gilles Dutilh

See Also

Refine() for adding iterations or parameter combinations to exsiting power_array object, SummarizeIterations() for summarizing a power_array object containing individual iterations, ArraySlicer() and [.power_array for reducing the dimenstiona of a power_array object, correctly updating its attributes.

Examples

```
## =======================================================
## most basic use case, calculating power when
## power function is available:
## =======================================================

## Define grid of assumptions to study:
sse_pars = list(
  n = seq(from = 10, to = 60, by = 2),         # sample size
  delta = seq(from = 0.5, to = 1.5, by = 0.2), # effect size
  sd = seq(.1, .9, .2))                        # standard deviation

## Define function that calculates power based on these assumptions:
PowFun <- function(n, delta, sd){
  ptt = power.t.test(n = n/2, delta = delta, sd = sd,
                     sig.level = 0.05)
  return(ptt$power)
}

## Evaluate at each combination of assumptions:
```

```
powarr = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = NA)
summary(powarr)

## ================================
## Use powergrid utilities on result
## ================================

## get required sample size n, when delta is .7, sd = .5, for achieving a
## power of 90%:
Example(powarr, example = list(delta = .7, sd = .5), target_value = .9)

## Draw a figure illustrating how the required n depends on delta (given an
## sd of .7):
PowerPlot(powarr,
          slicer = list(sd = .7), # slice out the plane with sd = .7
          target_value = .9, # set target power to 90%, defining the thick line
          example = list(delta = .7) # Highlight the example with arrow
          )
## Slice out a sub-array (making sure attributes stay intact for further use in
## powergrid):

only_n20_delta1.1 =
  ArraySlicer(powarr, slicer = list(
                        n = 20,
                        delta = 1.1))
summary(only_n20_delta1.1)

## Indexing may also be used, but note that the name of the remaining dimension
## is lost. Therefore, use ArraySlicer when you want to keep working with the
## object in powergrid.
only_n20_delta1.1 = powarr[n = 20, delta = 1.1, ]
summary(only_n20_delta1.1)

## =======================================================
## Simulation over iterations when no power
## function is available
## =======================================================

## Using the same assumptions as above
sse_pars = list(
  n = seq(from = 10, to = 60, by = 5),
  delta = seq(from = 0.5, to = 1.5, by = 0.2),
  sd = seq(.5, 1.5, .2))

## Define a function that results in TRUE or FALSE for a successful or
## non-successful (5% significant) simulated trial:
PowFun <- function(n, delta, sd){
  x1 = rnorm(n = n/2, sd = sd)
  x2 = rnorm(n = n/2, mean = delta, sd = sd)
  t.test(x1, x2)$p.value < .05
}

## In call to PowerGrid, setting n_iter prompts PowerGrid to evaluate
```

```
## the function iteratively at each combination of assumptions:
n_iter = 20
powarr = PowerGrid(pars = sse_pars, fun = PowFun,
                        n_iter = n_iter)

## By default, the iterations are summarized (by their mean), so:
dimnames(powarr)
summary(powarr) # indicates that iterations were summarized (not stored)

## ================================
## keeping individual iterations
## ================================

## To keep individual iterations, set summarize to FALSE:

powarr_no_summary = PowerGrid(pars = sse_pars, fun = PowFun,
                                    n_iter = n_iter , summarize = FALSE)
dimnames(powarr_no_summary) # additional dimension "iter"
summary(powarr_no_summary)

## To summarize this object containing iterations, use the SummarizeIterations
## function. Among other things, this assures that attributes relevant for
## further use in powergrid's functionality are kept intact.

powarr_summarized =
  SummarizeIterations(powarr_no_summary, summary_function = mean)
dimnames(powarr_summarized)
summary(powarr_summarized)

## This summarized `power_array` is no different from a version that was
## directly summarized.

## Note that Example and Powerplot detect when a `power_array` object is not
#summarized, and behave sensibly with a warning:
Example(powarr_no_summary, example = list(delta = .7, sd = .5), target_value = .9)

PowerPlot(powarr_no_summary,
          slicer = list(sd = .7), # slice out the plane with sd = .7
          target_value = .9, # set target power to 90%, defining the thick line
          example = list(delta = .7) # Highlight the example with arrow
          )

#=======================================================
# Multiple outputs are automatically handled #
#=======================================================

## Parameter assumptions
sse_pars = list(
  n = seq(from = 10, to = 60, by = 2),
  delta = seq(from = 0.5, to = 1.5, by = 0.2),
  sd = seq(.5, 1.5, .2))

## A function with two outputs (the power at two significance levels)
```

```
TwoValuesFun <- function(n, delta, sd){
  p5 = power.t.test(n = n, delta = delta, sd = sd, sig.level = .05)$power
  p1 = power.t.test(n = n, delta = delta, sd = sd, sig.level = .01)$power
  return(c('p5' = p5, 'p1' = p1))
}

powarr_two_returns = PowerGrid(sse_pars, TwoValuesFun)

## multiple outputs result in an additional dimension:
dimnames(powarr_two_returns)
summary(powarr_two_returns)

## note that you need to tell Example and other powergrid functions, which
## of the outputs you are interested in:
Example(powarr_two_returns, example = list(delta = .7, sd = .5, fun_out = 'p1'),
        target_value = .9)

PowerPlot(powarr_two_returns,
          slicer = list(sd = .7, fun_out = 'p1'), # slice out the plane with the
                                                  # output of interest
          target_value = .9, # set target power to 90%, defining the thick line
          example = list(delta = .7) # Highlight the example with arrow
          )
```

PowerPlot	*Plot the relation between assumed parameters and requirements for achieving a target power (or other objective)*

Description

Plot (a slice of) an object of class power_array. Main purpose is to illustrate the relation between two parameters (e.g., effect size on the x-axis and n on the y-axis) for a given target power. An example may be highlighted by drawing an arrow at the combination of parameters deemed most likely.

Usage

```
PowerPlot(
  x,
  slicer = NULL,
  par_to_search = "n",
  example = NULL,
  find_lowest = TRUE,
  target_value = 0.9,
  target_at_least = TRUE,
  method = "step",
  summary_function = mean,
  target_levels = c(0.8, 0.9, 0.95),
  col = grDevices::grey.colors(1, 0.2, 0.2),
```

```
  shades_of_grey = TRUE,
  example_text = TRUE,
  title = NULL,
  par_labels = NULL,
  smooth = NA,
  ...
)
```

Arguments

`x` An object of class `power_array` (from powergrid).

`slicer` If the parameter grid for which 'x' was constructed has more than 2 dimensions, a 2-dimensional slice may be cut out using `slicer`, which is a list whose elements define at which values (the list element value) of which parameter (the list element name) the slice should be cut out.

`par_to_search` The variable whose minimum (or maximum, when `find_lowest == FALSE`) is searched for achieving the `target_levels`.

`example` If not NULL, a list of length one, defining at which value (list element value) of which parameter (list element name) the example is drawn for a power of `target_value`. You may supply a vector longer than 1 for multiple examples.

`find_lowest` Logical, indicating whether the example should be found that minimizes an assumption (e.g., minimal required n) to achieve the `target_value` or an example that maximizes this assumption (e.g., maximally allowed SD).

`target_value` The power (or whatever the target is) for which the example, if requested, is drawn. Also defines which of the power lines is drawn with a thicker line width, among or in addition to the power lines defined by target_levels.

`target_at_least`
Logical. Should the target value be minimally achieved (e.g., power), or maximially allowed (e.g., estimation uncertainty).

`method` Method used for finding the required `par_to_search` needed to achieve `target_value`. Either `step`: walking in steps along `par_to_search` or `lm`: Interpolating assuming a linear relation between `par_to_search` and `(qnorm(x) + qnorm(1 - 0.05)) ^ 2`. The setting `lm` is inspired on the implementation in the `sse` package by Thomas Fabbro.

`summary_function`
If `x` is an object of class `power_array` where attribute `summarized` is FALSE (and individual iterations are stored in dimension `iter`, the iterations dimension is aggregated by `summary_fun`. Otherwise ignored.

`target_levels` For which levels of power (or whichever variable is contained in `x`) lines are drawn.

`col` Color for the contour lines. Does not effect eventual example arrows. Therefore, use AddExample.

`shades_of_grey` Logical indicating whether greylevels are painted in addition to isolines to show power levels.

`example_text` When an example is drawn, should the the required par value be printed alongside the arrow(s)

- `title` Character string, if not `NULL`, replaces default figure title.
- `par_labels` Named vector with elements named as the parameters plotted, with as values the desired labels.
- `smooth` Numeric, defaults to NA, meaning no smoothing. Non NA value is used as argument `span` for smoothing with `stats::loess`, regressing the contour values on the x and y-axis. Suggested value is .35. Functionality implemented for consistency with `sse` package, but use is discouraged, since regressing the contour values flattens the contour plot, thereby *biasing* the contour lines.
- `...` Further arguments are passed on to function `image` internally. Most useful for zooming with xlim and ylim.

Details

The most common use case may be plotting the required n (on the y-axis) as a function of some other parameter (e.g., effect size, on the x-axis) for achieving a certain level of statistical power. The default argument settings reflect this use case.

Flexible plotting:

The plotting is, however, more flexible.

Any variable on the axes:
You can flip the axes by setting a different `par_to_search` (which defines the y-axis). The other parameter is automatically chosen to be drawn on the x-axis.

Maximizing a parameter:
One may also search not the minimum, as in the case of sample size, but the maximum, e.g., the highest sd at which a certain power may still be achieved. In this case, the `par_to_search` is sd, and `find_lowest = FALSE`.

When smaller is better:
In the standard case of power, higher is better, so you search for a *minimal* level of power. One may however also aim at, e.g., a *maximal* width of a confidence interval. For this purpose, set `target_at_least` to `FALSE`. See Example for more details about `find_lowest` and `target_at_least`.

Value

A list containing the coordinate arguments x, y, and z, as passed to `image()` internally.

Author(s)

Gilles Dutilh

See Also

`PowerGrid`, `AddExample`, `Example`, `GridPlot` for plotting interdependencies of 3 parameters.

Examples

```
## ============================================
## Typical use case: minimal n for power
## ============================================
## What's the minimal sample size n, given the combination of sd and delta.

## Set up a grid of n, delta and sd:
sse_pars = list(
  n = seq(from = 10, to = 60, by = 4),
  delta = seq(from = 0.5, to = 1.5, by = 0.1), # effect size
  sd = seq(.1, 1.1, .2)) # Standard deviation

## Define a power function using these parameters:
PowFun <- function(n, delta, sd){ # power for a t-test at alpha = .05
  ptt = power.t.test(n = n/2, delta = delta, sd = sd,
                     sig.level = 0.05)
  return(ptt$power)
}

## Evaluate PowFun across the grid defined by sse_pars:
power_array = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = NA)

## explore power graphically in the situation where sd = .7, including an
## example situation where delta is .9:
PowerPlot(power_array,
          slicer = list(sd = .7),
          example = list(delta = c(.7, .9)), # two examples
          target_value = .9 # 90% power
          )

## Some graphical adjustments. Note that example is drawn on top of
## PowerPlot now.
PowerPlot(power_array,
          slicer = list(sd = .7),
          par_labels = c(n = 'Total Sample Size',
                         delta = 'Effect Size',
                         sd = 'Standard Deviation'),
          target_levels = c(.8, .9), # draw fewer power isolines
          target_value = NA # no specific power target (no line thicker)
          )
AddExample(power_array,
           slicer = list(sd = .7),
           example = list(delta = .9),
           target_value = .9,
           col = 'Orange', lwd = 3)

## ============================================
## Less typical use case:
## minimal delta for power, given sd, as a function of n
## ============================================
## You can easily change what you search for. For example: At each sample size n,
## what would be the minimal effect size delta there must be for the target
```

```
## power to be achieved?

PowerPlot(power_array,
          par_to_search = 'delta',
          slicer = list(sd = .7))

## ===============================================
## Less typical use case:
## *maximum sd* for power, given n, as a function of delta
## ===============================================
## You're not limited to study n at all, nor to searching a minimum: When
## your n is given to be 30, what is the largest sd at which we still find
## enough power? (as a function of delta on the x-axis)

PowerPlot(power_array,
          par_to_search = 'sd',
          find_lowest = FALSE,
          slicer = list(n = 30))

## Adding an example works the same: If we expect a delta of 1, and the n =
## 30, what is the maximal SD we can have still yielding 90% power?

AddExample(power_array,
           find_lowest = FALSE,
           slicer = list(n = 30),
           example = list(delta = 1),
           target_value = .9)
```

---

print.power_array *print*

Description

Method for printing objects of class power_array. ##' Prints a power_array as a default array with a short summary about its contents.

Usage

```
## S3 method for class 'power_array'
print(x, ...)
```

Arguments

x	object of class power_array
...	passed on to cat

Value

Nothing

Author(s)

Gilles Dutilh

See Also

PowerGrid

Examples

```
## Define grid of assumptions to study:
sse_pars = list(
  n = seq(from = 10, to = 50, by = 20),         # sample size
  delta = seq(from = 0.5, to = 1.5, by = 0.5), # effect size
  sd = seq(.1, 1, .3))                        # standard deviation

## Define function that calculates power based on these assumptions:
PowFun <- function(n, delta, sd){
  ptt = power.t.test(n = n/2, delta = delta, sd = sd,
                     sig.level = 0.05)
  return(ptt$power)
}

## Evaluate at each combination of assumptions:
powarr = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = NA)
print(powarr)
```

---

print.power_example *Print example*

Description

Print method for class power_example.

Usage

```
## S3 method for class 'power_example'
print(x, ...)
```

Arguments

x	object of class power_example
...	passed on to cat

Details

Print short informative output for object of class power_example.

Value

nothing

Author(s)

Gilles Dutilh

Refine *Refine or extend the result of PowerGrid*

Description

Add further results to an existing power_array (created by PowerGrid or by another call of Refine), adding further values in `pars` and/or larger `n_iter`.

Usage

```
Refine(old, n_iter_add = 1, pars = NULL, ...)
```

Arguments

`old`	the object of class `power_array` to extend
`n_iter_add`	the number of iterations to *add* to old
`pars`	the new parameter grid to evaluate across
`...`	further arguments passed on to PowerGrid internally.

Details

If `pars` == `NULL`, update extends `old` by adding iterations `n_iter_add` to the existing power_array. If `pars` is given, the function that was evaluated in `old` (attribute `sim_function`) is evaluated at the crossings of `pars`. If argument `pars` is different from `attr(old, which = 'pars')`, this means that the function is evaluated additional crossings of parameters.

Note that certain combinations of `pars` and `n_iter_add` result in arrays where some crossings of parameters include more iterations than others. This is a feature, not a bug. May result in less aesthetic plotting, however.

For details about handling the random seed, see `PowerGrid`.

Value

object of class `power_array`, containing old, extended by `pars` and/or `n_iter_add`.

Author(s)

Gilles Dutilh

See Also

PowerGrid

Examples

```
## ============================================
## very simple example with one parameter
## ============================================
pars = list(x = 1:2)
fun = function(x){round(x+runif(1, 0, .2), 3)} # nonsense function
set.seed(1)
original = PowerGrid(pars = pars,
                     fun = fun,
                     n_iter = 3,
                     summarize = FALSE)
refined = Refine(original, n_iter_add = 2, pars = list(x = 2:3))
## note that refined does not have each parameter sampled in each iteration

## ============================================
## a realistic example, simply increasing n_iter
## ============================================
PowFun <- function(n, delta){
  x1 = rnorm(n = n/2, sd = 1)
  x2 = rnorm(n = n/2, mean = delta, sd = 1)
  t.test(x1, x2)$p.value < .05
}
sse_pars = list(
  n = seq(10, 100, 5),
  delta = seq(.5, 1.5, .1))
##
n_iter = 20
set.seed(1)
power_array = PowerGrid(pars = sse_pars,
                        fun = PowFun,
                        n_iter = n_iter,
                        summarize = FALSE)
summary(power_array)
## add iterations
power_array_up = Refine(power_array, n_iter_add = 30)
summary(power_array_up)

## ============================================
## Starting coarsely, then zooming in
## ============================================
sse_pars = list(
  n = c(10, 50, 100, 200), # finding n "ballpark"
  delta = c(.5,  1, 1.5)) # finding delta "ballpark"
n_iter = 60
power_array = PowerGrid(pars = sse_pars,
                        fun = PowFun,
                        n_iter = n_iter,
                        summarize = FALSE)
```

```
summary(power_array)
PowerPlot(power_array)
## Based on figure above, let's look at n between 50 and 100, delta around .9

sse_pars = list(
  n = seq(50, 100, 5),
  delta = seq(.7, 1.1, .05))
set.seed(1)
power_array_up = Refine(power_array, n_iter_add = 555, pars = sse_pars)
summary(power_array_up)
PowerPlot(power_array_up) # that looks funny! It's because the default summary
                          # mean does not deal with the empty value in the
                          # grid. Solution is in illustration below.

## A visual illustration of this zooming in, in three figures
layout(t(1:3))
PowerPlot(power_array, title = 'Course grid to start with')
PowerPlot(power_array_up, summary_function = function(x)mean(x, na.rm = TRUE),
          title = 'Extra samples at finer parameter grid (does not look good)')
PowerPlot(power_array_up,
          slicer = list(n = seq(50, 100, 5),
                        delta = seq(.7, 1.1, .05)),
          summary_function = function(x)mean(x, na.rm = TRUE),
          title = 'Zoomed in')
layout(1)
```

SummarizeIterations *Summary of object that has individual iterations saved.*

Description

Summarizes objects of class `power_array` that have individual iterations saved.

Usage

```
SummarizeIterations(x, summary_function, ...)
```

Arguments

`x`	Object of class power_array
`summary_function`	function to apply across iterations
`...`	Further arguments passed to 'summary_function'

Value

An object of class `power_array`, with attributes `summarized = TRUE`.

Author(s)

Gilles Dutilh

See Also

PowerGrid

Examples

```
## iterative sse example
sse_pars = list(
  n = seq(from = 10, to = 60, by = 5),
  delta = seq(from = 0.5, to = 1.5, by = 0.2),
  sd = seq(.5, 1.5, .2))

## Define a function that results in TRUE or FALSE for a successful or
## non-successful (5% significant) simulated trial:
PowFun <- function(n, delta, sd){
  x1 = rnorm(n = n/2, sd = sd)
  x2 = rnorm(n = n/2, mean = delta, sd = sd)
  t.test(x1, x2)$p.value < .05
}

n_iter = 20
powarr = PowerGrid(pars = sse_pars, fun = PowFun,
                        n_iter = n_iter, summarize = FALSE)

dimnames(powarr)
summary(powarr) # indicates that iterations were not
## now summarize
powarr_summarized = SummarizeIterations(powarr, summary_function = mean)
dimnames(powarr_summarized)
summary(powarr_summarized) # indicates that iterations are now summarized
```

---

summary.power_array *Summary of power_grid object.*

Description

Offers a short summary of the power_array object, summarizing the range of observed values and the grid evaluated across. ##' See PowerGrid for details

Usage

```
## S3 method for class 'power_array'
summary(object, ...)
```

Arguments

`object`	array of class power_grid
`...`	passed on to `cat`

Value

nothing

Author(s)

Gilles Dutilh

See Also

PowerGrid

Examples

```
## Define grid of assumptions to study:
sse_pars = list(
  n = seq(from = 10, to = 50, by = 20),         # sample size
  delta = seq(from = 0.5, to = 1.5, by = 0.5), # effect size
  sd = seq(.1, 1, .3))                        # standard deviation

## Define function that calculates power based on these assumptions:
PowFun <- function(n, delta, sd){
  ptt = power.t.test(n = n/2, delta = delta, sd = sd,
                   sig.level = 0.05)
  return(ptt$power)
}

## Evaluate at each combination of assumptions:
powarr = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = NA)
summary(powarr)
```

---

summary.power_example *Print contents of an example*

Description

Summary method for class `power_example`.

Usage

```
## S3 method for class 'power_example'
summary(object, ...)
```

Arguments

`object`	object of class `power_example`
`...`	passed on to `data.frame` (which is the thing that is printed)

Details

Print longer informative output for object of class `power_example`.

Value

nothing

Author(s)

Gilles Dutilh

`[.power_array` *indexing with [ ] for class* `power_array` *[ ]: R:%20*

Description

Method for indexing [] of objects of class power_array. The method makes sure that the resulting array is of class power_array and keeps and updates the object's attributes. These attributes are needed for various functions in the powergrid package to work well. ##' The indexing functions as normal indexing, but note that drop is FALSE by default, so that the resulting array has the same dimensions as the original array. The number of levels at each dimension may be reduced, however. ##'

Usage

```
## S3 method for class 'power_array'
x[..., drop = TRUE]
```

Arguments

`x`	object
`...`	index
`drop`	drop

Value

An array of class `power_grid`

Author(s)

Gilles Dutilh

See Also

PowerGrid ArraySlicer for a different method of reducing the dimensions of an array of class power_array.

Examples

```
## Define grid of assumptions to study:
sse_pars = list(
  n = seq(from = 10, to = 50, by = 20),         # sample size
  delta = seq(from = 0.5, to = 1.5, by = 0.5), # effect size
  sd = seq(.1, 1, .3))                         # standard deviation

## Define function that calculates power based on these assumptions:
PowFun <- function(n, delta, sd){
  ptt = power.t.test(n = n/2, delta = delta, sd = sd,
                     sig.level = 0.05)
  return(ptt$power)
}

## Evaluate at each combination of assumptions:
powarr = PowerGrid(pars = sse_pars, fun = PowFun, n_iter = NA)
powarr[2, 1, ] # gives the same as
powarr['30', '0.5', ]
```

Index