Package: power4mome (via r-universe)

June 8, 2026

Title Power Analysis for Moderation and Mediation

Version 0.2.1.3

Description Power analysis and sample size determination for moderation, mediation, and moderated mediation in models fitted by structural equation modelling using the 'lavaan' package by Rosseel (2012) <doi:10.18637/jss.v048.i02> or by multiple regression. The package 'manymome' by Cheung and Cheung (2024) <doi:10.3758/s13428-023-02224-z> is used to specify the indirect paths or conditional indirect paths to be tested.

License GPL (>= 3)

Encoding UTF-8

Roxygen list(markdown = TRUE)

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

Config/testthat/parallel true

Depends R (>= 4.4.0)

URL https://sfcheung.github.io/power4mome/

BugReports https://github.com/sfcheung/power4mome/issues

Imports lavaan, stats, manymome (>= 0.3.1.4), pbapply, parallel, pgnorm, lmhelprs (>= 0.4.2), psych, yaml, graphics, methods, cli, mice

Config/Needs/website rmarkdown

LazyData true

VignetteBuilder knitr

Config/roxygen2/version 8.0.0

Config/pak/sysreqs cmake libglpk-dev make libicu-dev libxml2-dev libx11-dev zlib1g-dev

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

Date/Publication 2026-05-31 12:42:12 UTC

RemoteUrl https://github.com/sfcheung/power4mome

RemoteRef HEAD

RemoteSha 13b469d7977dd52f37a8bf66f42404dd21baf86d

Contents

bz_helpers *Helpers for the Boos-and-Zhang (2000) Method*

Description

Helpers for the method by Boos and Zhang (2000) for estimating rejection rate for resampling-based tests.

Usage

```
Rs_bz_supported(
  alpha = 0.05,
  Rmax = getOption("power4mome.bz_Rmax", default = 359)
)

R_for_bz(R_target, alpha = 0.05)
```

Arguments

alpha	The level of significance, two-tailed.
Rmax	The maximum number of resamples to be returned. Default is 359. Though it is possible to use 1999 resamples or even more with Boos-Zhang-2000, using such a large number of resamples defeats the goal to reduce processing time.
R_target	The target maximum number of resamples.

Details

Boos and Zhang (2000) proposed a method to estimate the rejection rate (power, if the null hypothesis is false) for methods based on resampling, such as nonparametric bootstrapping. This method is used by some functions in power4mome, such as `x_from_power()`.

This method is implemented internally. Some helper functions regarding this method is exported for users.

The function `Rs_bz_supported()` returns the number of bootstrap samples (for bootstrapping) or simulated samples (for Monte Carlo), both called resamples below for brevity, usually specified by the argument R, that can be used for the Boos-Zhang-2000 method, given the desired two-tailed level of significance, (.05 by default). If possible, setting the number of resamples. (e.g., setting R when calling `x_from_power()`) will automatically enable the Boos-Zhang-2000 method (unless explicitly turned off by setting the option "power4mome.bz" to FALSE by `options("power4mome.bz") <- FALSE`), substantially reducing the processing time. For now, only two-tailed tests are supported.

Given a target maximum number of resamples and a level of significance, the function `R_for_bz()` returns largest number of resamples that can be used for the Boos-Zhang-2000 method. This function can be used for arguments such as R in `x_from_power()` to automatically find the largest value supported by the Boos-Zhang-2000 method.

Value

The function `Rs_bz_supported()` returns a numeric vector of the numbers of resamples supported.

The function `R_for_bz()` returns a scalar.

References

Boos, D. D., & Zhang, J. (2000). Monte Carlo evaluation of resampling-based hypothesis tests. *Journal of the American Statistical Association*, *95*(450), 486–492. doi:10.1080/01621459.2000.10474226

See Also

`x_from_power()`

Examples

```
# === Rs_bz_supported ===

# alpha = .05
Rs_bz_supported()

# alpha = .01
Rs_bz_supported(alpha = .01)

# === R_for_bz ===

R_for_bz(200)
R_for_bz(500)
```

---

do_test *Do a Test on Each Replication*

Description

Do a test on each replication in the output of `sim_out()`.

Usage

```
do_test(
  sim_all,
  test_fun,
  test_args = list(),
  map_names = c(fit = "fit"),
  results_fun = NULL,
  results_args = list(),
  parallel = FALSE,
  progress = FALSE,
  ncores = max(1, parallel::detectCores(logical = FALSE) - 1),
```

```
  cl = NULL
)
```

Arguments

sim_all	The output of sim_out().
test_fun	A function to do the test. See 'Details' for the requirement of this function. There are some built-in test functions in power4mome, described in 'Details'.
test_args	A list of arguments to be passed to the test_fun function. Default is list().
map_names	A named character vector specifying how the content of the element extra in each replication of sim_all map to the argument of test_fun. Default is c(fit = "fit"), indicating that the element fit in the element extra is set to the argument fit of test_fun. That is, for the first replication, fit = sim_out[[1]]$extra$fit when calling test_fun.
results_fun	The function to be used to extract the test results. See Details for the requirements of this function. Default is NULL, assuming that the output of test_fun can be used directly.
results_args	A list of arguments to be passed to the results_fun function. Default is list().
parallel	If TRUE, parallel processing will be used to do the tests. Default is FALSE.
progress	If TRUE, the progress of tests will be displayed. Default is 'FALSE.
ncores	The number of CPU cores to use if parallel processing is used.
cl	A cluster, such as one created by parallel::makeCluster(). If NULL, a cluster will be created, but will be stopped on exit. If set to an existing cluster, it will not be stopped when the function exits; users need to stop it manually.

Details

The function do_test() does an arbitrary test in each replication using the function set to test_fun.

Value

An object of the class test_out, which is a list of length equal to sim_all, one element for each replication. Each element of the list has two elements:

- test: The output of the function set to test_fun.
- test_results: The output of the function set to results_fun.

The role of do_test()

The function do_test() is used by the all-in-one function power4test(). Users usually do not call this function directly, though developers can use this function to develop other functions for power analysis, or to build their own workflows to do the power analysis.

Major Test-Related Arguments

The test function (test_fun):

The function set to `test_fun`, the *test function*, usually should work on the output of `lavaan::sem()`, `lmhelprs::many_lm()`, or `stats::lm()`, but can also be a function that works on the output of the function set to `fit_function` when calling `fit_model()` or `power4test()` (see `fit_model_args`).

The function has two default requirements.

First, it has an argument `fit`, to be set to the output of `lavaan::sem()` or another output stored in the element `extra$fit` of a replication in the `sim_all` object. (This requirement can be relaxed; see the section on `map_names`.)

That is, the function definition should be of this from: `FUN(fit, ...)`. This is the form of all `test_*` functions in `power4mome`.

If other arguments are to be passed to the test function, they can be set to `test_args` as a named list.

Second, the test function must returns an output that (a) can be processed by the results function (see below), or (b) is of the required format for the output of a results function (see the next section). If it already returns an output of the required format, then there is no need to set the results function.

The results function (results_fun):

The test results will be extracted from the output of `test_fun` by the function set to `results_fun`, the *results function*. If the `test_fun` already returns an output of the expected format (see below), then set `results_fun` to `NULL`, the default. The output of `test_fun` will be used for estimating power.

The function set to `results_fun` must accept the output of `test_fun`, as the first argument, and return a named list (which can be a data frame) or a named vector with some of the following elements:

- `est`: Optional. The estimate of a parameter, if applicable.
- `se`: Optional. The standard error of the estimate, if applicable.
- `cilo`: Optional. The lower limit of the confidence interval, if applicable.
- `cihi`: Optional. The upper limit of the confidence interval, if applicable.
- `sig`: Required. If `1`, the test is significant. If `0`, the test is not significant. If the test cannot be done for any reason, it should be `NA`.

The results can then be used to estimate the power or Type I error of the test.

For example, if the null hypothesis is false, then the proportion of significant, that is, the mean of the values of `sig` across replications, is the power.

Built-in test functions:

The package `power4mome` has some ready-to-use test functions:

- `test_indirect_effect()`
- `test_cond_indirect()`
- `test_cond_indirect_effects()`
- `test_moderation()`
- `test_index_of_mome()`
- `test_parameters()`

Please refer to their help pages for examples.

The argument map_names:

This argument is for developers using a test function that has a different name for the argument of the fit object ("fit", the default).

If test_fun is set to a function that works on an output of, say, lavaan::sem() but the argument name for the output is not fit, the mapping can be changed by map_names.

For example, lavaan::parameterEstimates() receives an output of lavaan::sem() and reports the test results of model parameters. However, the argument name for the lavaan output is object. To instruct do_test() to do the test correctly when setting test_fun to lavaan::parameterEstimates, add map_names = c(object = "fit"). The element fit in a replication will then set to the argument object of lavaan::parameterEstimates().

The background for having the results_fun argument

In the early development of power4mome, test_fun is designed to accept existing functions from other packages, such as manymome::indirect_effect(). Their outputs are not of the required format for power analysis, and so results functions are needed to process their outputs. In the current version of power4mome, some ready-to-use test functions, usually wrappers of these existing functions from other packages, have been developed, and they no longer need results functions to process the output. The argument results_fun is kept for backward compatibility and advanced users to use test functions from other packages.

See Also

See power4test() for the all-in-one function that uses this function. See test_indirect_effect(), test_cond_indirect(), test_cond_indirect_effects(), test_moderation(), test_index_of_mome(), and test_parameters() for examples of test functions.

Examples

```
# Specify the population model

mod <-
"
m ~ x
y ~ m + x
"


# Specify the effect sizes (population parameter values)

es <-
"
y ~ m: m
m ~ x: m
y ~ x: n
"


# Generate several simulated datasets

data_all <- sim_data(nrep = 5,
```

```
                      model = mod,
                      pop_es = es,
                      n = 100,
                      iseed = 1234)

# Fit the population model to each datasets

fit_all <- fit_model(data_all)

# Generate Monte Carlo estimates for forming Monte Carlo confidence intervals

mc_all <- gen_mc(fit_all,
                 R = 50,
                 iseed = 4567)

# Combine the results to a 'sim_all' object
sim_all <- sim_out(data_all = data_all,
                   fit = fit_all,
                   mc_out = mc_all)

# Test the indirect effect in each replication
# Set `parallel` to TRUE for faster, usually much faster, analysis
# Set `progress` to TRUE to display the progress of the analysis

test_all <- do_test(sim_all,
                    test_fun = test_indirect_effect,
                    test_args = list(x = "x",
                                     m = "m",
                                     y = "y",
                                     mc_ci = TRUE),
                    parallel = FALSE,
                    progress = FALSE)

# The result for each dataset
lapply(test_all, function(x) x$test_results)
```

`fit_model` *Fit a Model to a List of Datasets*

Description

Get the output of `sim_data()` and fit a model to each of the stored datasets.

Usage

```
fit_model(
  data_all = NULL,
  model = NULL,
```

```
  fit_function = "lavaan",
  arg_data_name = "data",
  arg_model_name = "model",
  arg_group_name = "group",
  ...,
  fit_out = NULL,
  parallel = FALSE,
  progress = FALSE,
  ncores = max(1, parallel::detectCores(logical = FALSE) - 1),
  cl = NULL
)
```

Arguments

data_all	The output of sim_data(), or a sim_data class object.
model	The model to be fitted. If NULL, the default, the model stored in data_all, which should be the data generation model, will be used.
fit_function	The function to be used to fit the model. Can also be a string: "lavaan" (the default) for lavaan::sem(), and "lm" or many_lm for lmhelprs::many_lm(). Other functions can also be used if necessary.
arg_data_name	The name of the argument of fit_function expecting the dataset. Default is "data".
arg_model_name	The name of the argument of fit_function expecting the model definition. Default is "model".
arg_group_name	The name of the argument of fit_function expecting the name of the group variable. Used only for multigroup models. Default is "group".
...	Optional arguments to be passed to fit_function when fitting the model.
fit_out	If set to a fit_out object (the output of fit_model()), then all missing arguments will be retrieved from fit_out. That is, users can use fit_model(data_all = new_data, fit_out = old_out) to re-fit a model originally fitted in old_out on a new list of dataset (new_data). No need to include all other arguments.
parallel	If TRUE, parallel processing will be used to fit the models. Default is FALSE.
progress	If TRUE, the progress of model fitting will be displayed. Default is 'FALSE.
ncores	The number of CPU cores to use if parallel processing is used.
cl	A cluster, such as one created by parallel::makeCluster(). If NULL, a cluster will be created, but will be stopped on exit. If set to an existing cluster, it will not be stopped when the function exits; users need to stop it manually.

Details

By default, the function fit_model()

- extracts the model stored in the output of sim_data(),
- fits the model to each dataset simulated using fit_function, default to "lavaan" and lavaan::sem() will be called,

- and returns the results.

If the datasets were generated from a multigroup model when calling `sim_data()`, a multigroup model is fitted.

Value

An object of the class `fit_out`, which is a list of the output of `fit_function` (`lavaan::sem()` by default). If an error occurred when fitting the model to a dataset, then this element will be the error message from the fit function.

The role of `fit_model()`

This function is used by the all-in-one function `power4test()`. Users usually do not call this function directly, though developers can use this function to customize the model fitting step in power analysis.

See Also

See `power4test()` for the all-in-one function that uses this function, and `sim_data()` for the function generating datasets for this function.

Examples

```
# Specify the population model

mod <-
"m ~ x
 y ~ m + x"

# Specify the effect sizes (population parameter values)

es <-
"
y ~ m: m
m ~ x: m
y ~ x: n
"

# Generate several simulated datasets

data_all <- sim_data(nrep = 5,
                     model = mod,
                     pop_es = es,
                     n = 100,
                     iseed = 1234)

# Fit the population model to each datasets

fit_all <- fit_model(data_all)
fit_all[[1]]
```

```
# Fit the population model using the MLR estimator

fit_all_mlr <- fit_model(data_all,
                         estimator = "MLR")
fit_all_mlr[[1]]

# Fit a model different from the population model,
# with the MLR estimator

mod2 <-
"m ~ x
 y ~ m"

fit_all_mlr2 <- fit_model(data_all,
                          mod2,
                          estimator = "MLR")
fit_all_mlr2[[1]]
```

---

gen_boot *Generate Bootstrap Estimates*

Description

Get a list of the output of `lavaan::sem()` or `lmhelprs::many_lm()` and generate bootstrap estimates of model parameters.

Usage

```
gen_boot(
  fit_all,
  R = 100,
  ...,
  iseed = NULL,
  parallel = FALSE,
  progress = FALSE,
  ncores = max(1, parallel::detectCores(logical = FALSE) - 1),
  compute_implied_stats = FALSE,
  cl = NULL
)
```

Arguments

`fit_all`	The output of `fit_model()` or an object of the class `fit_out`.
`R`	The number of replications to generate the bootstrap estimates for each fit output.
`...`	Optional arguments to be passed to `manymome::do_boot()` when generating the bootstrap estimates.

iseed	The seed for the random number generator. Default is NULL and the seed is not changed.
parallel	If TRUE, parallel processing will be used to generate bootstrap estimates for the fit outputs. Default is FALSE.
progress	If TRUE, the progress will be displayed. Default is 'FALSE.
ncores	The number of CPU cores to use if parallel processing is used.
compute_implied_stats	Whether implied statistics are computed in each bootstrap samples. Usually not needed and so default to FALSE
cl	A cluster, such as one created by parallel::makeCluster(). If NULL, a cluster will be created, but will be stopped on exit. If set to an existing cluster, it will not be stopped when the function exits; users need to stop it manually.

Details

The function gen_boot() simply calls manymome::do_boot() on each output of lavaan::sem() or lmhelprs::many_lm() in fit_all. The simulated estimates can then be used to test effects such as indirect effects, usually by functions from the manymome package, such as manymome::indirect_effect().

Value

A boot_list object, which is a list of the output of manymome::do_boot().

The role of gen_boot()

This function is used by the all-in-one function power4test(). Users usually do not call this function directly, though developers can use this function to customize the workflow of the power analysis.

See Also

See power4test() for the all-in-one function that uses this function.

Examples

```
# Specify the population model

mod <-
"m ~ x
 y ~ m + x"

# Specify the effect sizes (population parameter values)

es <-
"
y ~ m: m
m ~ x: m
y ~ x: n
"
```

```
# Generate several simulated datasets

data_all <- sim_data(nrep = 2,
                     model = mod,
                     pop_es = es,
                     n = 50,
                     iseed = 1234)

# Fit the population model to each datasets

fit_all <- fit_model(data_all)

# Generate bootstrap estimates for each replication

boot_all <- gen_boot(fit_all,
                     R = 10,
                     iseed = 4567)
boot_all
```

---

gen_mc *Generate Monte Carlo Estimates*

Description

Get a list of the output of `lavaan::sem()` and generate Monte Carlo estimates of model parameters.

Usage

```
gen_mc(
  fit_all,
  R = 100,
  ...,
  iseed = NULL,
  parallel = FALSE,
  progress = FALSE,
  ncores = max(1, parallel::detectCores(logical = FALSE) - 1),
  compute_implied_stats = FALSE,
  cl = NULL
)
```

Arguments

`fit_all`	The output of `fit_model()` or an object of the class `fit_out`.
`R`	The number of replications to generate the Monte Carlo estimates for each fit output.

...	Optional arguments to be passed to manymome::do_mc() when generating the Monte Carlo estimates.
iseed	The seed for the random number generator. Default is NULL and the seed is not changed.
parallel	If TRUE, parallel processing will be used to generate Monte Carlo estimates for the fit outputs. Default is FALSE.
progress	If TRUE, the progress will be displayed. Default is 'FALSE.
ncores	The number of CPU cores to use if parallel processing is used.
compute_implied_stats	Whether implied statistics are computed in each Monte Carlo replication. Usually not needed and so default to FALSE.
cl	A cluster, such as one created by parallel::makeCluster(). If NULL, a cluster will be created, but will be stopped on exit. If set to an existing cluster, it will not be stopped when the function exits; users need to stop it manually.

Details

The function gen_mc() simply calls manymome::do_mc() on each output of lavaan::sem() in fit_all. The simulated estimates can then be used to test effects such as indirect effects, usually by functions from the manymome package, such as manymome::indirect_effect().

Value

An mc_list object, which is a list of the output of manymome::do_mc().

The role of gen_mc()

This function is used by the all-in-one function power4test(). Users usually do not call this function directly, though developers can use this function to customize the workflow of the power analysis.

See Also

See power4test() for the all-in-one function that uses this function.

Examples

```
# Specify the population model

mod <-
"m ~ x
 y ~ m + x"

# Specify the effect sizes (population parameter values)

es <-
"
y ~ m: m
m ~ x: m
```

```
y ~ x: n
"

# Generate several simulated datasets

data_all <- sim_data(nrep = 5,
                     model = mod,
                     pop_es = es,
                     n = 100,
                     iseed = 1234)

# Fit the population model to each datasets

fit_all <- fit_model(data_all)

# Generate Monte Carlo estimates for each replication

mc_all <- gen_mc(fit_all,
                 R = 100,
                 iseed = 4567)

mc_all
```

`missing_values` *Process Data by Generating Missing Values*

Description

For the `process_data` argument. It introduces missing values in the generated data.

Usage

```
missing_values(data, ..., prop = 0.5, mech = "MCAR")
```

Arguments

`data`	A data frame.
`...`	Optional arguments to be passed to `mice::ampute()`.
`prop`	The proportion of missingness. Default is 0.5, about 50% of the cases have missing data.
`mech`	The missing data mechanism. Default is `"MCAR"` (missing completely at random). Other possible values are `"MAR"` (missing at random) and `"MNAR"` (missing not at random). Please refer to the help of `mice::ampute()` for details.

Details

This function is to be used in the process_data argument of power4test().

The missing values are generated by mice::ampute(). Please refer to its help page, the vignette for this function: Generate missing values with ampute, and Schouten, Lugtig, and Vink (2018).

In addition to data, only two arguments are explicitly defined for missing_values(): prop and mech. The argument prop is defined to remind users of the default value for this argument. The argument mech, specifying the missing data mechanism, is defined because its default value is different from that of mice::ampute(). For missing_values(), the default value is "MCAR", missing completely at random.

Please refer to the help page of mice::ampute() for other available arguments.

Note for patterns and other arguments with variable names:

If arguments such as patterns are specified, the variable names need to be those in the generated data. If indicator scores are generated, the names need to be the names of the indicators, not the names of the latent variables.

Value

It returns a data frame with the missing values inserted.

References

Schouten, R. M., Lugtig, P., & Vink, G. (2018). Generating missing values for simulation purposes: A multivariate amputation procedure. *Journal of Statistical Computation and Simulation, 88*(15), 2909–2930. doi:10.1080/00949655.2018.1491577

See Also

power4test(). See also mice::ampute() for the amputation method.

Examples

```
# Specify the model

mod <-
"
m ~ x
y ~ m + x
"

# Specify the population values

mod_es <-
"
y ~ m: l
m ~ x: m
y ~ x: n
"
# Simulate the data
```

```
out <- power4test(
         nrep = 2,
         model = mod,
         pop_es = mod_es,
         n = 200,
         process_data = list(fun = "missing_values",
                             args = list(prop = .75)),
         test_fun = test_parameters,
         test_args = list(op = "~"),
         parallel = FALSE,
         iseed = 1234)

dat <- pool_sim_data(out)
head(dat, 50)
```

`ordinal_variables` *Process Data by Creating Ordinal Variables*

Description

For the `process_data` argument. It converts continuous indicator variable to ordinal variables.

Usage

```
ordinal_variables(data, cut_patterns = NULL, cuts = NULL)

cut_patterns(which = NULL)
```

Arguments

`data`	A data frame.
`cut_patterns`	A named vector. The names are the names of the latent variables for which indicator scores will be converted. Each value must be the name of one of the built-in patterns (call `cut_patterns()` to list the patterns and their names). Can be used with `cuts` but a latent variable should appear only either in `cut_patterns` or `cuts`, not both.
`cuts`	A named list. The names are the names of the latent variables for which indicator scores will be converted. Each element is a vector of the thresholds for the conversion. `-Inf` and `Inf` will be automatically included during the conversion. Can be used with `cut_patterns` but a latent variable should appear only either in `cut_patterns` or `cuts`, not both.
`which`	A name of a built-in pattern.

Details

This function is to be used in the `process_data` argument of `power4test()`.

It is used for converting the continuous indicator scores generated to ordinal scores (with values 1, 2, 3, etc.).

For example, if the cut points (thresholds) are -2 and 2, then sores will be converted to three categories: (-Inf to -2], (-2 to 2], and (2 to Inf]. The intervals are closed on the right. That is, a score of -2 is in the interval (-Inf to -2], not in the interval (-2 to 2).

The conversion is implemented by `cut()`.

There are two ways to specify the conversion. The first one is to use some built-in thresholds, based on Savalei and Rhemtulla (2013), Table 1. Call `cut_patterns()` with no argument to list all the built-in patterns and their names.

Alternatively, users can specify the thresholds directly through the argument `cuts`.

Currently, all indicators of a latent variable must be converted in the same way.

Value

The function `ordinal_variables()` returns a data frame with the the converted scores.

The function `cut_patterns()` returns a named list of the built-in patterns if `which = NULL`, and a numeric vector of the thresholds of a built-in pattern if `which` is set to one of the names of the built-in patterns.

References

Savalei, V., & Rhemtulla, M. (2013). The performance of robust test statistics with categorical data. *British Journal of Mathematical and Statistical Psychology*, *66*(2), 201–223. doi:10.1111/j.20448317.2012.02049.x

See Also

`power4test()`. See also `cut()` for the implementation.

Examples

```
# Specify the model

mod <-
"
m ~ x
y ~ m + x
"

# Specify the population values

mod_es <-
"
y ~ m: l
m ~ x: m
y ~ x: n
```

```
"

# Specify the numbers of indicators and reliability coefficients

k <- c(y = 3,
       m = 4,
       x = 5)
rel <- c(y = .70,
         m = .70,
         x = .70)

# Simulate the data

out <- power4test(
         nrep = 2,
         model = mod,
         pop_es = mod_es,
         n = 200,
         number_of_indicators = k,
         reliability = rel,
         process_data = list(fun = "ordinal_variables",
                             args = list(cut_patterns = c(x = "s3"),
                                         cuts = list(m = c(-2, 0, 2)))),
         test_fun = test_parameters,
         test_args = list(op = "~"),
         parallel = FALSE,
         iseed = 1234)

dat <- pool_sim_data(out)
head(dat, 50)

# The built-in patterns

cut_patterns()
```

`plot.power_curve` *Plot a Power Curve*

Description

Plotting the results in a 'power_curve' object, such as the estimated power against sample size, or the results of `power4test_by_n()` or `power4test_by_es()`.

Usage

```
## S3 method for class 'power_curve'
plot(
  x,
  what = c("ci", "power_curve"),
```

```
 main = paste0("Power Curve ", "(Predictor: ", switch(x$predictor, n = "Sample Size", es
   = "Effect Size"), ")"),
  xlab = switch(x$predictor, n = "Sample Size", es = "Effect Size"),
  ylab = "Estimated Power",
  pars_ci = list(),
  type = "l",
  ylim = c(0, 1),
  ci_level = 0.95,
  ...
)

## S3 method for class 'power4test_by_n'
plot(
  x,
  what = c("ci", "power_curve"),
  main = "Estimated Power vs. Sample Size",
  xlab = "Sample Size",
  ylab = "Estimated Power",
  pars_ci = list(),
  type = "l",
  ylim = c(0, 1),
  ci_level = 0.95,
  ...
)

## S3 method for class 'power4test_by_es'
plot(
  x,
  what = c("ci", "power_curve"),
  main = paste0("Estimated Power vs. Effect Size / Parameter (", attr(x[[1]],
    "pop_es_name"), ")"),
  xlab = paste0("Effect Size / Parameter (", attr(x[[1]], "pop_es_name"), ")"),
  ylab = "Estiamted Power",
  pars_ci = list(),
  type = "l",
  ylim = c(0, 1),
  ci_level = 0.95,
  ...
)
```

Arguments

x	The object to be plotted. It can be a power_curve object, the output of power_curve(). It can also be the output of power4test_by_n() or power4test_by_es().
what	A character vector of what to include in the plot. Possible values are "ci" (confidence intervals for the estimated sample size) and "power_curve" (the crude power curve, if available). The default values depend on the type of x.
main	The title of the plot.

xlab, ylab	The labels for the horizontal and vertical axes, respectively.
pars_ci	A named list of arguments to be passed to arrows() to customize the drawing of the confidence intervals.
type	An argument of the default plot method plot.default(). Default is "l". See plot.default() for other options.
ylim	A two-element numeric vector of the range of the vertical axis.
ci_level	The level of confidence of the confidence intervals, if requested. Default is .95, denoting 95%.
...	Optional arguments. Passed to plot() when drawing the base plot.

Details

The plot method of power_curve objects currently plots the relation between estimated power and the predictor. Other elements can be requested (see the argument what), and they can be customized individually.

Value

The plot-methods return x invisibly. They are called for their side effects.

See Also

power_curve(), power4test_by_n(), and power4test_by_es().

Examples

```
# Specify the population model

model_simple_med <-
"
m ~ x
y ~ m + x
"


# Specify the effect sizes (population parameter values)

model_simple_med_es <-
"
y ~ m: l
m ~ x: m
y ~ x: s
"


# Simulate datasets to check the model

sim_only <- power4test(nrep = 10,
                       model = model_simple_med,
                       pop_es = model_simple_med_es,
                       n = 50,
                       fit_model_args = list(fit_function = "lm"),
```

```
                        do_the_test = FALSE,
                        iseed = 1234,
                        parallel = FALSE,
                        progress = FALSE)

# By n: Do a test for different sample sizes
# Set `parallel` to TRUE for faster, usually much faster, analysis
# Set `progress` to TRUE to display the progress of the analysis

out1 <- power4test_by_n(sim_only,
                         nrep = 10,
                         test_fun = test_parameters,
                         test_args = list(par = "y~x"),
                         n = c(25, 50, 100),
                         by_seed = 1234,
                         parallel = FALSE,
                         progress = FALSE)

pout1 <- power_curve(out1)
pout1
plot(pout1)

# By pop_es: Do a test for different population values of a model parameter
# Set `parallel` to TRUE for faster, usually much faster, analysis
# Set `progress` to TRUE to display the progress of the analysis

out2 <- power4test_by_es(sim_only,
                             nrep = 10,
                             test_fun = test_parameters,
                             test_args = list(par = "y~x"),
                             pop_es_name = "y ~ x",
                             pop_es_values = c(0, .3, .5),
                             by_seed = 1234,
                             parallel = FALSE,
                             progress = FALSE)

pout2 <- power_curve(out2)
plot(pout2)
```

`plot.x_from_power` *Plot The Results of 'x_from_power'*

Description

It plots the results of 'x_from_power', such as the estimated power against sample size.

Usage

```
## S3 method for class 'x_from_power'
```

```
plot(
  x,
  what = c("ci", "power_curve", "final_x", "final_power", "target_power", switch(x$x, n =
    "sig_area", es = NULL)),
  text_what = c("final_x", "final_power", switch(x$x, n = "sig_area", es = NULL)),
  digits = 3,
  main = NULL,
  xlab = NULL,
  ylab = "Estimated Power",
  ci_level = 0.95,
  pars_ci = list(),
  pars_power_curve = list(),
  pars_ci_final_x = list(lwd = 3, length = 0.2, col = "blue"),
  pars_target_power = list(lty = "dashed", lwd = 2, col = "black"),
  pars_final_x = list(lty = "dotted"),
  pars_final_power = list(lty = "dotted", col = "blue"),
  pars_text_final_x = list(y = 0, pos = 3, cex = 1),
  pars_text_final_power = list(pos = 3, cex = 1),
  pars_sig_area = list(col = adjustcolor("lightblue", alpha.f = 0.1)),
  pars_text_sig_area = list(cex = 1),
  prop_of_trials = NULL,
  min_trials_for_prop = NULL,
  override_for_pba = TRUE,
  ...
)

## S3 method for class 'n_region_from_power'
plot(
  x,
  what = c("ci", "power_curve", "final_x", "final_power", "target_power", "sig_area"),
  text_what = c("final_x", "final_power", "sig_area"),
  digits = 3,
  main = paste0("Power Curve ", "(Target Power: ", formatC(x$below$target_power, digits =
    digits, format = "f"), ")"),
  xlab = NULL,
  ylab = "Estimated Power",
  ci_level = 0.95,
  pars_ci = list(),
  pars_power_curve = list(),
  pars_ci_final_x = list(lwd = 2, length = 0.2, col = "blue"),
  pars_target_power = list(lty = "dashed", lwd = 2, col = "black"),
  pars_final_x = list(lty = "dotted"),
  pars_final_power = list(lty = "dotted", col = "blue"),
  pars_text_final_x = list(pos = 3, cex = 1),
  pars_text_final_x_lower = pars_text_final_x,
  pars_text_final_x_upper = pars_text_final_x,
  pars_text_final_power = list(cex = 1),
  pars_sig_area = list(col = adjustcolor("lightblue", alpha.f = 0.1)),
```

```
  pars_text_sig_area = list(cex = 1),
  ...
)
```

Arguments

x	An x_from_power object, the output of x_from_power().
what	A character vector of what to include in the plot. Possible values are "ci" (confidence intervals for the estimated value of the predictor), "power_curve" (the crude power curve, if available), "final_x" (a vertical line for the value of the predictor with estimated power close enough to the target power by confidence interval), "final_power" (a horizontal line for the estimated power of the final value of the predictor), "target_power" (a horizontal line for the target power), and "sig_area" (the area significantly higher or lower than the target power, if goal is "close_enough" and what is "lb" or "ub"). By default, all these elements will be plotted.
text_what	A character vector of what numbers to be added as labels. Possible values are "final_x" (the value of the predictor with estimated power close enough to the target power by confidence interval) "final_power" (the estimated power of the final value of the predictor), and "sig_area" (labeling the area significantly higher or lower than the target power, if goal is "close_enough" and what is "lb" or "ub"). By default, all these labels will be added.
digits	The number of digits after the decimal that will be used when adding numbers.
main	The title of the plot. If NULL, it will be generated automatically.
xlab, ylab	The labels for the horizontal and vertical axes, respectively.
ci_level	The level of confidence of the confidence intervals, if requested. Default is .95, denoting 95%.
pars_ci	A named list of arguments to be passed to arrows() to customize the drawing of the confidence intervals.
pars_power_curve	A named list of arguments to be passed to points() to customize the drawing of the power curve.
pars_ci_final_x	A named list of arguments to be passed to arrows() to customize the drawing of the confidence interval of the final value of the predictor.
pars_target_power	A named list of arguments to be passed to abline() when drawing the horizontal line for the target power.
pars_final_x	A named list of arguments to be passed to abline() when drawing the vertical line for the final value of the predictor.
pars_final_power	A named list of arguments to be passed to abline() when drawing the horizontal line for the estimated power at the final value of the predictor.
pars_text_final_x	A named list of arguments to be passed to text() when adding the label for the final value of the predictor.

pars_text_final_power
: A named list of arguments to be passed to text() when adding the label for the estimated power of final value of the predictor.

pars_sig_area
: A named list of arguments to be passed to rect() when shading the area significantly higher or lower than the target power.

pars_text_sig_area
: A named list of arguments to be passed to text() when labelling the area significantly higher or lower than the target power.

prop_of_trials
: The proportion of trials to be included in the plot. If NULL, it will be determined based on the algorithm used.

min_trials_for_prop
: The minimum number of trials for prop_of_trials to be used. If NULL, it will be determined based on the algorithm used.

override_for_pba
: If TRUE, the default, the values of some arguments will be overriden internally if the algorithm used is "probabilistic_bisection", to make the plot suitable for this algorithm. For example, the power curve and the confidence intervals for trials other than the solution will not be plotted, even if requested. If FALSE, then argument values will be not be changed internally.

...
: Optional arguments. Passed to plot() when drawing the estimated power against the predictor.

pars_text_final_x_lower, pars_text_final_x_upper
: If two values of the predictor are to be printed, these are the named list of the arguments to be passed to text() when adding the labels for these two values.

Details

The plot method of x_from_power objects currently plots the relation between estimated power and the values examined by x_from_power(). Other elements can be requested (see the argument what), and they can be customized individually.

The plot-method for n_region_from_power objects is a modified version of the plot-method for x_from_power. It plots the results of two runs of n_from_power() in one plot. It is otherwise similar to the plot-method for x_from_power.

Value

The plot-method of x_from_power returns x invisibly. It is called for its side effect.

The plot-method of n_region_from_power returns x invisibly. It is called for its side effect.

See Also

x_from_power()

Examples

```
# Specify the population model
```

```
mod <-
"
m ~ x
y ~ m + x
"

# Specify the population values

mod_es <-
"
m ~ x: m
y ~ m: l
y ~ x: n
"

# Generate the datasets

sim_only <- power4test(nrep = 10,
                       model = mod,
                       pop_es = mod_es,
                       n = 100,
                       do_the_test = FALSE,
                       iseed = 1234)

# Do a test

test_out <- power4test(object = sim_only,
                       test_fun = test_parameters,
                       test_args = list(pars = "m~x"))

# Determine the sample size with a power of .80 (default)

power_vs_n <- x_from_power(test_out,
                           x = "n",
                           progress = TRUE,
                           target_power = .80,
                           final_nrep = 10,
                           max_trials = 1,
                           seed = 2345)
plot(power_vs_n)
```

---

pop_es_yaml *Parse YAML-Stye Values For 'pop_es'*

Description

Convert a YAML string to a vector or list of pop_es specification.

Usage

```
pop_es_yaml(text)
```

Arguments

text	The multiline string to be parsed to a specification of population values.

Details

The function pop_es_yaml() allows users to specify population values of a model using one single string, as in 'lavaan' model syntax.

Value

Either a named vector (for a single-group model) or a list of named vector (for a multigroup model) of the population values of the parameters (the effect sizes).

Specify 'pop_es' Using a Multiline String

When setting the argument pop_es, instead of using a named vector or named list for pop_es, the population values of model parameters can also be specified using a multiline string, as illustrated below, to be parsed by pop_es_yaml().

Single-Group Model:

This is an example of the multiline string for a single-group model:

```
y ~ m: l
m ~ x: m
y ~ x: nil
```

The string must follow this format:

- Each line starts with tag:.
  - tag can be the name of a parameter, in lavaan model syntax format.
    - For example, m ~ x denotes the path from x to m.
  - A tag in lavaan model syntax can specify more than one parameter using +.
    - For example, y ~ m + x denotes the two paths from m and x to y.
  - Alternatively, the tag can be either .beta. or .cov..
    - Use .beta. to set the default values for all regression coefficients.
    - Use .cov. to set the default values for all correlations of exogenous variables (e.g., predictors).
- After each tag is the value of the population value:
  -nil for nil (zero),
  - s for small,
  - m for medium, and
  - l for large.
  - si, mi, and li for small, medium, and large a standardized indirect effect, respectively.

  Note: n *cannot* be used in this mode.
  The value for each label is determined by es1 and es2 as described in ptable_pop().

- The value can also be set to a numeric value, such as `.30` or `-.30`.

This is another example:

```
.beta.: s
y ~ m: l
```

In this example, all regression coefficients are `small`, while the path from `m` to `y` is large.

Multigroup Model:

This is an example of the string for a multigroup model:

```
y ~ m: l
m ~ x:
  - nil
  - s
y ~ x: nil
```

The format is similar to that for a single-group model. If a parameter has the same value for all groups, then the line can be specified as in the case of a single-group model: `tag: value`.

If a parameter has different values across groups, then it must be in this format:

- A line starts with the tag, followed by two or more lines. Each line starts with a hyphen `-` and the value for a group.

For example:

```
m ~ x:
  - nil
  - s
```

This denotes that the model has two groups. The values of the path from `x` to `m` for the two groups are 0 (`nil`) and small (`s`), respectively.

Another equivalent way to specify the values are using `[]`, on the same line of a tag.

For example:

```
m ~ x: [nil, s]
```

The number of groups is inferred from the number of values for a parameter. Therefore, if a tag has more than one value, each tag must has the same number of value, or only one value.

The tag `.beta.` and `.cov.` can also be used for multigroup models.

Which Approach To Use:

Note that using named vectors or named lists is more reliable. However, using a multiline string is more user-friendly. If this method failed, please use named vectors or named list instead.

Technical Details:

The multiline string is parsed by `yaml::read_yaml()`. Therefore, the format requirement is actually that of YAML. Users knowledgeable of YAML can use other equivalent way to specify the string.

See Also

`ptable_pop()`, `power4test()`, and other functions that have the `pop_es` argument.

Examples

```
mod_es <- c("y ~ m" = "l",
            "m ~ x" = "m",
            "y ~ x" = "nil")

mod_es_yaml <-
"
y ~ m: l
m ~ x: m
y ~ x: nil
"

pop_es_yaml(mod_es_yaml)
```

`power_curve` *Power Curve*

Description

Estimate the relation between power and a characteristic, such as sample size or population effect size (population value of a model parameter).

Usage

```
power_curve(
  object,
  formula = NULL,
  start = NULL,
  lower_bound = NULL,
  upper_bound = NULL,
  nls_args = list(),
  nls_control = list(),
  verbose = FALSE,
  models = c("nls", "logistic", "lm"),
  nls_options = list(min_points = 4, regions = c(0, 0.45, 0.75, 0.85, 1))
)

## S3 method for class 'power_curve'
print(x, data_used = FALSE, digits = 3, right = FALSE, row.names = FALSE, ...)
```

Arguments

`object`	An object of the class `power4test_by_n` or `power4test_by_es`, which is the output of `power4test_by_n()` or `power4test_by_es()`.

formula — A formula of the model for stats::nls(). It can also be a list of formulas, and the models will be fitted successively by stats::nls(), with the first model fitted successfully adopted. The response variable in the formula must be named reject, and the predictor named x. Whether x represents n or es depends on the class of object. If NULL, the default, it will be determined internally based on the type of object.

start — Either a named vector of the start value(s) of parameter(s) in formula, or a list of named vectors of the starting value(s) of the list of formula(s). If NULL, the default, they will be determined internally.

lower_bound — Either a named vector of the lower bound(s) of parameter(s) in formula, or a list of named vectors of the lower bound(s) for the list of formula(s). They will be passed to lower of stats::nls(). If NULL, the default, it will be determined internally based on the type of object.

upper_bound — Either a named vector of the upper bound(s) of parameter(s) in formula, or a list of named vectors of the upper bound(s) for the list of formula(s). They will be passed to upper of stats::nls(). If NULL, the default, it will be determined internally based on the type of object.

nls_args — A named list of arguments to be used when calling stats::nls(). Used to override internal default, such as the algorithm (default is "port"). Use this argument with cautions.

nls_control — A named list of arguments to be passed the control argument of stats::nls(). The values will override internal default values, and also override nls_args. Use this argument with cautions.

verbose — Logical. Whether messages will be printed when trying different models.

models — Models to try. Support "nls" (fitted by nls()), "logistic" (fitted by glm()), and "lm" (fitted by lm()). By default, all three models will be attempted, in this order.

nls_options — A named list of options to be used by power_curve() to configure the use of "nls". For advanced use. See 'Details' for available options.

x — A power_curve object.

data_used — Logical. Whether the rejection rates data frame used to fit the model is printed.

digits, right, row.names — Arguments of the same names used by the print method of a data.frame object. Used when data_used is TRUE and the rejection rates data frame is printed.

... — For the print method of power_curve objects, they are optional arguments to be passed to print.data.frame() when printing the rejection rates data frame.

Details

The function power_curve() retrieves the information from the output of power4test_by_n() or power4test_by_es(), and estimate the power curve: the relation between the characteristic varied, sample size for power4test_by_n() and the population effect size for power4test_by_es(), and the rejection rate of the test conducted by power4test_by_n() or power4test_by_es(). This rejection rate is the power when the null hypothesis is false (e.g., the population value of the effect size being tested is nonzero).

The model fitted is *not* intended to be a precise model for the relation across a wide range. It is only a crude estimate based on the limited number of values of the characteristic (e.g., sample size) examined, which can be as small as four or even smaller. The model is intended to be used for only for the range covered, and for estimating the probable sample size or effect size with a desirable level of power. This value should then be studied by higher precision through simulation using functions such as `power4test()`.

These are the models to be tried, in the following order:

- One or nonlinear models, to be fitted by `stats::nls()`. If several models are specified, all will be fitted and the one with the smallest deviance will be used.
- If all the nonlinear models failed, for whatever reason, a logistic regression will be fitted by `stats::glm()` to predict the binary significant test results.
- If the logistic model also failed, for whatever reason, a simple linear regression model will be fitted. Although the power curve is nonlinear across a wide range of, say, sample size, a linear model can still be a good enough approximation for a narrow range of the predictor.

The output can then be plotted to visualize the power curve, using the `plot` method (`plot.power_curve()`) for the output of `power_curve()`.

This function can be used directly, but is also used internally by functions such as `x_from_power()`.

'nls_options':

These are possible arguments.

- `min_points`: If the object has less than `min_points` rejection rates, `nls` will not be used.
- `regions`: A numeric vector with values from 0 to 1. It defines regions in which there must be at least one rejection rate. If this criterion is not met, `nls` will not be used. This is to ensure that `nls` will not be used when the rejection rates are clustered around a very narrow region.

Value

It returns a list which is a `power_curve` object, with the following elements:

- `fit`: The model fitted, which is the output of `stats::nls()`, `stats::glm()`, or `stats::lm()`.
- `reject_df`: The table of reject rates and other characteristics, which is generated by `rejection_rates()`.
- `predictor`: The predictor or the power curve, ether `"n"` (sample size) or `"es"` (population effect size).
- `call`: The call used to run this function.

The `print` method of `power_curve` object returns `x` invisibly. Called for its side-effect.

See Also

`power4test_by_n()` and `power4test_by_es()` for the output supported by `power_curve()`, `plot.power_curve()` for the `plot` method and `predict.power_curve()` for the `predict` method of the output of `power_curve()`.

Examples

```
# Specify the population model

model_simple_med <-
"
m ~ x
y ~ m + x
"

# Specify the effect sizes (population parameter values)

model_simple_med_es <-
"
y ~ m: l
m ~ x: m
y ~ x: s
"

# Simulate datasets to check the model
# Set `parallel` to TRUE for faster, usually much faster, analysis
# Set `progress` to TRUE to display the progress of the analysis

sim_only <- power4test(nrep = 10,
                       model = model_simple_med,
                       pop_es = model_simple_med_es,
                       n = 50,
                       fit_model_args = list(fit_function = "lm"),
                       do_the_test = FALSE,
                       iseed = 1234,
                       parallel = FALSE,
                       progress = FALSE)

# By n: Do a test for different sample sizes

out1 <- power4test_by_n(sim_only,
                        nrep = 10,
                        test_fun = test_parameters,
                        test_args = list(par = "y~x"),
                        n = c(25, 50, 100),
                        by_seed = 1234,
                        parallel = FALSE,
                        progress = FALSE)

pout1 <- power_curve(out1)
pout1
plot(pout1)

# By pop_es: Do a test for different population values of a model parameter

out2 <- power4test_by_es(sim_only,
                         nrep = 10,
                         test_fun = test_parameters,
```

```
                          test_args = list(par = "y~x"),
                          pop_es_name = "y ~ x",
                          pop_es_values = c(0, .3, .5),
                          by_seed = 1234,
                          parallel = FALSE,
                          progress = FALSE)

pout2 <- power_curve(out2)
pout2
plot(pout2)
```

---

power4test *Estimate the Power of a Test*

Description

An all-in-one function that receives a model specification, generates datasets, fits a model, does the target test, and returns the test results.

Usage

```
power4test(
  object = NULL,
  nrep = NULL,
  ptable = NULL,
  model = NULL,
  pop_es = NULL,
  standardized = TRUE,
  n = NULL,
  number_of_indicators = NULL,
  reliability = NULL,
  loading_difference = NULL,
  reference = NULL,
  x_fun = list(),
  e_fun = list(),
  process_data = NULL,
  fit_model_args = list(),
  R = NULL,
  ci_type = "mc",
  gen_mc_args = list(),
  gen_boot_args = list(),
  test_fun = NULL,
  test_args = list(),
  map_names = c(fit = "fit"),
  results_fun = NULL,
  results_args = list(),
  test_name = NULL,
```

```
  test_note = NULL,
  do_the_test = TRUE,
  sim_all = NULL,
  iseed = NULL,
  parallel = FALSE,
  progress = TRUE,
  ncores = max(1, parallel::detectCores(logical = FALSE) - 1),
 es1 = c(n = 0, nil = 0, s = 0.1, m = 0.3, l = 0.5, si = 0.141, mi = 0.361, li = 0.51,
    sm = 0.2, ml = 0.4),
  es2 = c(n = 0, nil = 0, s = 0.05, m = 0.1, l = 0.15, sm = 0.075, ml = 0.125),
  es_ind = c("si", "mi", "li"),
  n_std = 1e+05,
  std_force_monte_carlo = FALSE,
 rejection_rates_args = list(collapse = "none", at_least_k = 1, merge_all_tests = FALSE,
    p_adjust_method = "none", alpha = 0.05),
  n_ratio = 1
)

## S3 method for class 'power4test'
print(
  x,
  what = c("data", "test"),
  digits = 3,
  digits_descriptive = 2,
  data_long = FALSE,
  test_long = TRUE,
  fit_to_all_args = list(),
  ...
)
```

Arguments

object	Optional. If set to a power4test object, it will be updated using the value(s) in n, pop_es, and/or nrep if they changed, or a new test will be conducted and added to objet. See the help page for details. Default is NULL.
nrep	The number of replications to generate the simulated datasets. Default is NULL. Must be set when called to create a power4test object.
ptable	The output of ptable_pop(), which is a ptable_pop object, representing the population model. If NULL, the default, ptable_pop() will be called to generate the ptable_pop object using model and pop_es.
model	The lavaan model syntax of the population model, to be used by ptable_pop(). See 'Details' of on how to specify the model. Ignored if ptable is specified.
pop_es	The character vector or multiline string to specify population effect sizes (population values of parameters). See the help page on how to specify this argument. Ignored if ptable is specified.
standardized	Logical. If TRUE, the default, variances and error variances are scaled to ensure the population variances of the endogenous variables are close to one, and hence

the effect sizes (population values) are standardized effect sizes if the variances of the continuous exogenous variables are also equal to one.

`n` The sample size for each dataset. Default is 100.

`number_of_indicators`
A named vector to specify the number of indicators for each factors. See the help page on how to set this argument. Default is `NULL` and all variables in the model syntax are observed variables. See the help page on how to use this argument.

`reliability` A named vector (for a single-group model) or a named list of named vectors (for a multigroup model) to set the reliability coefficient of each set of indicators. Default is `NULL`. See the help page on how to use this argument.

`loading_difference`
A named vector (for a single-group model) or a named list of named vectors (for a multigroup model) to set the difference in factor loadings between neighboring indicators of each set of indicators. Default is `NULL`, and all indicators of a factor have the same factor loadings. If specified, must be specified for all factors named in `reliability`, even for those with all loadings equal.

`reference` A named vector (for a single-group model) or a named list of named vectors (for a multigroup model) to indicate which indicator will be the first indicator (and so is the reference indicator, by default). Default is `NULL`, and for all factors, the indicator with the medium loading in a factor is the first indicator. Has no effect if loading difference is zero (and so all indicators have the same loadings). If specified, must be specified for all factors named in `reliability`, even for those with all loadings equal. Accepted values are `"medium"`, `"weakest"`, and `"strongest"`.

`x_fun` The function(s) used to generate the exogenous variables or error terms. If not supplied, or set to `list()`, the default, the variables are generated from a multivariate normal distribution. See the help page on how to use this argument.

`e_fun` The function(s) used to generate the error terms of indicators, if any. If not supplied, or set to `list()`, the default, the error terms of indicators are generated from a multivariate normal distribution. Specify in the same way as `x_fun`. Refer to the help page on `x_fun` on how to use this argument.

`process_data` If not `NULL`, it must be a named list with these elements: `fun` (required), the function to further process the simulated data, such as generating missing data using functions such as `mice::ampute()`; `args` (optional), a named list of arguments to be passed to `fun`, except the one for the source data; `sim_data_name` (optional) the name of the argument to receive the simulated data (e.g., `"data"` for `mice::ampute()`), default to `"data"` if it is not set; `processed_data_name` (optional), the name of the data frame after being processed by `fun`, such as the data frame with missing data in the output of `fun` (e.g., `"amp"` for `mice::ampute()`), if omitted, the output of `fun` should be the data frame with missing data.

`fit_model_args` A list of the arguments to be passed to `fit_model()` when fitting the model. Should be a named list with names being the names of the arguments.

`R` The number of replications to generate the Monte Carlo or bootstrapping estimates for each fit output. No Monte Carlo nor bootstrapping estimates will be generated if `R` is set to `NULL`.

ci_type	The type of simulation-based confidence intervals to use. Can be either "mc" for Monte Carlo method (the default) or "boot" for nonparametric bootstrapping method. Relevant for test functions that make use of estimates generate by gen_boot() or gen_mc(), such as test_indirect_effect().
gen_mc_args	A list of arguments to be passed to manymome::do_mc() when generating the Monte Carlo estimates. Should be a named argument with names being the names of the arguments. Used only if ci_type is "mc".
gen_boot_args	A list of arguments to be passed to manymome::do_boot() when generating the bootstrap estimates. Should be a named argument with names being the names of the arguments. Used only if ci_type is '"boot".
test_fun	A function to do the test. See 'Details' for the requirement of this function. There are some built-in test functions in power4mome, described in 'Details'.
test_args	A list of arguments to be passed to the test_fun function. Default is list().
map_names	A named character vector specifying how the content of the element extra in each replication of sim_all map to the argument of test_fun. Default is c(fit = "fit"), indicating that the element fit in the element extra is set to the argument fit of test_fun. That is, for the first replication, fit = sim_out[[1]]$extra$fit when calling test_fun.
results_fun	The function to be used to extract the test results. See Details for the requirements of this function. Default is NULL, assuming that the output of test_fun can be used directly.
results_args	A list of arguments to be passed to the results_fun function. Default is list().
test_name	String. The name of the test. Default is NULL, and the name will be created from test_fun. Note that if sim_out is a power4test object and already has a test of this name stored, it will be replaced by the new results.
test_note	String. An optional note for the test, stored in the attribute test_note of the output of do_test(). Default is NULL.
do_the_test	If TRUE, do_test() will be called to do the test specified by test_fun on the fit output of each dataset.
sim_all	If set to either a sim_out object (the output of sim_out() or a power4test object (the output of power4test()), the stored datasets and fit outputs will be used for doing the test. Setting object to the output of power4test() is now the preferred method, but this argument is kept for backward compatibility.
iseed	The seed for the random number generator. Default is NULL and the seed is not changed. This seed will be set only once, when calling sim_data().
parallel	If TRUE, parallel processing will be used when calling other functions, if appropriate. Default is FALSE.
progress	If TRUE, the progress of each step will be displayed. Default is TRUE.
ncores	The number of CPU cores to use if parallel processing is used.
es1	Set the values for each label of the effect size (population value) for correlations and regression paths. Used only if pop_es is a named vector or a multiline string. See the help page on how to specify this argument.

es2 Set the values for each label of the effect size (population value) for product term. Used only if pop_es is a named vector or a multiline string. See the help page on how to specify this argument.

es_ind The names of labels denoting the effect size of an indirect effect. They will be used to determine the population values of the component paths along an indirect path.

n_std The sample size used to determine the error variances by simulation when std_force_monte_carlo is TRUE.

std_force_monte_carlo
Logical. If FALSE, the default, standardization is done analytically if the model has no product terms, and by simulation if the model has product terms. That is, error variances required to ensure implied variances equal to one are determined by simulation. If TRUE, simulation will be used whether the model has product terms or not. Always fall back to simulation if analytical standardization failed.

rejection_rates_args
Default argument values to be used when rejection_rates() is called. Can be overriden when calling rejection_rates().

n_ratio If the model is a multigroup model, and n is a single number, this should be a numeric vector used to determine the sample size for each group. For example, for a two-group model, if n is 100 and n_ratio is c(1, 0.5), then the sample sizes for the two groups are 100 and 50, respectively. If equal to 1, then all groups have the same sample size.

x The object to be printed.

what A string vector of what to print, "data" for simulated data and "test" for stored test(s). Default is c("data", "test").

digits The numbers of digits displayed after the decimal.

digits_descriptive
The number of digits displayed after the decimal for the descriptive statistics table.

data_long If TRUE, detailed results will be printed when printing the simulated data.

test_long If TRUE, detailed results will be printed when printing test(s).

fit_to_all_args
A named list of arguments to be passed to lavaan::sem() when the model is fitted to a sample combined from all samples stored.

... Optional arguments to be passed to other print methods

Details

The function power4test() is an all-in-one function for estimating the power of a test for a model, given the sample size and effect sizes (population values of model parameters).

Value

An object of the class power4test, which is a list with two elements:

- sim_all: The output of sim_out().

- test_all: A named list of the output of do_test(). The names are the values of test_name. This list can have more than one test because a call to power4test() can add new tests to a power4test object.

The print method of power4test returns x invisibly. Called for its side effect.

Workflow

This is the workflow:

- If object is an output of the output of a previous call to power4test() with do_the_test set to FALSE and so has only the model and the simulated data, the following steps will be skipped and go directly to doing the test.
  - Call sim_data() to determine the population model and generate the datasets, using arguments such as model and pop_es.
  - Call fit_model() to fit a model to each of the datasets, which is the population model by default.
  - If R is not NULL and ci_type = "mc", call gen_mc() to generate Monte Carlo estimates using manymome::do_mc(). The estimates can be used by supported functions such as test_indirect_effect().
  - If R is not NULL and ci_type = "boot", call gen_boot() to generate bootstrap estimates using manymome::do_boot(). The estimates can be used by supported functions such as test_indirect_effect().
  - Merge the results into a sim_out object by calling sim_out().
  - If do_the_test is FALSE, skip the remaining steps and return a power4test object, which contains only the data generated and optionally the Monte Carlo or bootstrap estimates.
- If do_the_test is TRUE, do the test.
  - do_test() will be called to do the test in the fit output of each dataset.
- Return a power4test object which include the output of sim_out and, if do_the_test is TRUE, the output of do_test().

This function is to be used when users are interested only in the power of one or several tests on a particular aspect of the model, such as a parameter, given a specific effect sizes and sample sizes.

Detailed description on major arguments can be found in sections below.

NOTE: The technical internal workflow of of power4test() can be found in this page: https://sfcheung.github.io/power4mome/articles/power4test_workflow.html.

Updating a Condition

The function power4test() can also be used to update a condition when only some selected aspects is to be changed.

For example, instead of calling this function with all the arguments set just to change the sample size, it can be called by supplying an existing power4test object and set only n to a new sample size. The data and the tests will be updated automatically. See the examples for an illustration.

Adding Another Test

The function power4test() can also be used to add a test to the output from a previous call to power4test().

For example, after simulating the datasets and doing one test, the output can be set to object of power4test(), and set only test_fun and, optionally, test_fun_args to do one more test on the generated datasets. The output will be the original object with the results of the new test added. See the examples for an illustration.

Model Fitting Arguments ('fit_model_args')

For power analysis, usually, the population model (model) is to be fitted, and there is no need to set fit_model_args.

If power analysis is to be conducted for fitting a model that is not the population model, of if non-default settings are desired when fitting a model, then the argument fit_model_args needed to be set to customize the call to fit_model().

For example, users may want to examine the power of a test when a misspecified model is fitted, or the power of a test when MLR is used as the estimator when calling lavaan::sem().

See the help page of fit_model() for some examples.

Specify the Population Model by 'model'

Single-Group Model:

For a single-group model, model should be a lavaan model syntax string of the *form* of the model. The population values of the model parameters are to be determined by pop_es.

If the model has latent factors, the syntax in model should specify only the *structural model* for the *latent factors*. There is no need to specify the measurement part. Other functions will generate the measurement part on top of this model.

For example, this is a simple mediation model:

```
"m ~ x
 y ~ m + x"
```

Whether m, x, and y denote observed variables or latent factors are determined by other functions, such as power4test().

Multigroup Model:

Because the model is the population model, equality constraints are irrelevant and the model syntax specifies only the *form* of the model. Therefore, model is specified as in the case of single group models.

Specify 'pop_es' Using Named Vectors

The argument pop_es is for specifying the population values of model parameters. This section describes how to do this using named vectors.

Single-Group Model:

If pop_es is specified by a named vector, it must follow the convention below.

- The names of the vectors are lavaan names for the selected parameters. For example, m ~ x denotes the path from x to m.
- Alternatively, the names can be either ".beta." or ".cov.". Use ".beta." to set the default values for all regression coefficients. Use ".cov." to set the default values for all correlations of exogenous variables (e.g., predictors).
- The names can also be of this form: ".ind.(<path>)", whether <path> denote path in the model. For example, ".ind.(x->m->y)" denotes the path from x through m to y. Alternatively, the lavaan symbol ~ can also be used: ".ind.(y~m~x)". This form is used to set the indirect effect (standardized, by default) along this path. The value for this name will override other settings.
- If using lavaan names, can specify more than one parameter using +. For example, y ~ m + x denotes the two paths from m and x to y.
- The value of each element can be the label for the effect size: n for nil, s for small, m for medium, and l for large. The value for each label is determined by es1 and es2. See the section on specifying these two arguments.
- The value of pop_es can also be set to a value, but it must be quoted as a string, such as "y ~ x" = ".31".

This is an example:

```
c(".beta." = "s",
  "m1 ~ x" = "-m",
  "m2 ~ m1" = "l",
  "y ~ x:w" = "s")
```

In this example,

- All regression coefficients are set to small (s) by default, unless specified otherwise.
- The path from x to m1 is set to medium and negative (-m).
- The path from m1 to m2 is set to large (l).
- The coefficient of the product term x:w when predicting y is set to small (s).

Indirect Effect:
When setting an indirect effect to a symbol (default: "si", "mi", "li", with "i" added to differentiate them from the labels for a direct path), the corresponding value is used to determine the population values of *all* component paths along the pathway. All the values are assumed to be equal. Therefore, ".ind.(x->m->y)" = ".20" is equivalent to setting m ~ x and y ~ m to the square root of .20, such that the corresponding indirect effect is equal to the designated value.
This behavior, though restricted, is for quick manipulation of the indirect effect. If different values along a pathway, set the value for each path directly.
Only nonnegative value is supported. Therefore, ".ind.(x->m->y)" = "-si" and ".ind.(x->m->y)" = "-.20" will throw an error.

Multigroup Model:

The argument pop_es also supports multigroup models.

For pop_es, instead of named vectors, named *list* of named vectors should be used.

- The names are the parameters, or keywords such as .beta. and .cov., like specifying the population values for a single group model.

- The elements are character vectors. If it has only one element (e.g., a single string), then it is the the population value for all groups. If it has more than one element (e.g., a vector of three strings), then they are the population values of the groups. For a model of k groups, each vector must have either k elements or one element.

This is an example:

```
list("m ~ x" = "m",
     "y ~ m" = c("s", "m", "l"))
```

In this model, the population value of the path `m ~ x` is medium (`m`) for all groups, while the population values for the path `y ~ m` are small (`s`), medium (`m`), and large (`l`), respectively.

Specify 'pop_es' Using a Multiline String

When setting the argument `pop_es`, instead of using a named vector or named list for `pop_es`, the population values of model parameters can also be specified using a multiline string, as illustrated below, to be parsed by `pop_es_yaml()`.

Single-Group Model:

This is an example of the multiline string for a single-group model:

```
y ~ m: l
m ~ x: m
y ~ x: nil
```

The string must follow this format:

- Each line starts with `tag:`.
  - `tag` can be the name of a parameter, in `lavaan` model syntax format.
    - For example, `m ~ x` denotes the path from `x` to `m`.
  - A tag in `lavaan` model syntax can specify more than one parameter using `+`.
    - For example, `y ~ m + x` denotes the two paths from `m` and `x` to `y`.
  - Alternatively, the `tag` can be either `.beta.` or `.cov.`.
    - Use `.beta.` to set the default values for all regression coefficients.
    - Use `.cov.` to set the default values for all correlations of exogenous variables (e.g., predictors).
- After each tag is the value of the population value:
  -`nil` for nil (zero),
  - `s` for small,
  - `m` for medium, and
  - `l` for large.
  - `si`, `mi`, and `li` for small, medium, and large a standardized indirect effect, respectively.

  Note: `n` *cannot* be used in this mode.
  The value for each label is determined by `es1` and `es2` as described in `ptable_pop()`.
  - The value can also be set to a numeric value, such as `.30` or `-.30`.

This is another example:

```
.beta.: s
y ~ m: l
```

In this example, all regression coefficients are `small`, while the path from `m` to `y` is large.

Multigroup Model:

This is an example of the string for a multigroup model:

```
y ~ m: l
m ~ x:
  - nil
  - s
y ~ x: nil
```

The format is similar to that for a single-group model. If a parameter has the same value for all groups, then the line can be specified as in the case of a single-group model: `tag: value`.

If a parameter has different values across groups, then it must be in this format:

- A line starts with the tag, followed by two or more lines. Each line starts with a hyphen `-` and the value for a group.

For example:

```
m ~ x:
  - nil
  - s
```

This denotes that the model has two groups. The values of the path from `x` to `m` for the two groups are 0 (`nil`) and small (`s`), respectively.

Another equivalent way to specify the values are using `[]`, on the same line of a tag.

For example:

```
m ~ x: [nil, s]
```

The number of groups is inferred from the number of values for a parameter. Therefore, if a tag has more than one value, each tag must has the same number of value, or only one value.

The tag `.beta.` and `.cov.` can also be used for multigroup models.

Which Approach To Use:

Note that using named vectors or named lists is more reliable. However, using a multiline string is more user-friendly. If this method failed, please use named vectors or named list instead.

Technical Details:

The multiline string is parsed by `yaml::read_yaml()`. Therefore, the format requirement is actually that of YAML. Users knowledgeable of YAML can use other equivalent way to specify the string.

Set the Values for Effect Size Labels ('es1' and 'es2')

The vector `es1` is for correlations, regression coefficients, and indirect effect, and the vector `es2` is for for standardized moderation effect, the coefficients of a product term. These labels are to be used in interpreting the specification in `pop_es`.

Set 'number_of_indicators' and 'reliability'

The arguments `number_of_indicators` and `reliability` are used to specify the number of indicators (e.g., items) for each factor, and the population reliability coefficient of each factor, if the variables in the model syntax are latent variables.

Optionally, `loading_difference` can be used to generate indicators with unequal standardized factor loadings, and `reference` can be used to specify the indicator with the medium, strongest, or weakest standardized factor loading as the first indicator, which is used as the reference indicator in `lavaan`.

Single-Group Model:

If a variable in the model is to be replaced by indicators in the generated data, set `number_of_indicators` to a named numeric vector. The names are the variables of variables with indicators, as appeared in the `model` syntax. The value of each name is the number of indicators.

The argument `reliability` should then be set a named numeric vector (or list, see the section on multigroup models) to specify the population reliability coefficient ("omega") of each set of indicators. The population standardized factor loadings are then computed to ensure that the population reliability coefficient is of the target value.

These are examples for a single group model:

```
number of indicator = c(m = 3, x = 4, y = 5)
```

The numbers of indicators for `m`, `x`, and `y` are 3, 4, and 5, respectively.

```
reliability = c(m = .90, x = .80, y = .70)
```

The population reliability coefficients of `m`, `x`, and `y` are .90, .80, and .70, respectively.

Multigroup Models:

Multigroup models are supported. The number of groups is inferred from `pop_es` (see the help page of `ptable_pop()`), or directly from `ptable`.

For a multigroup model, the number of indicators for each variable must be the same across groups.

However, the population reliability coefficients can be different across groups. For a multigroup model of k groups, with one or more population reliability coefficients differ across groups, the argument `reliability` should be set to a named list. The names are the variables to which the population reliability coefficients are to be set. The element for each name is either a single value for the common reliability coefficient, or a numeric vector of the reliability coefficient of each group.

This is an example of `reliability` for a model with 2 groups:

```
reliability = list(x = .80, m = c(.70, .80))
```

The reliability coefficients of `x` are .80 in all groups, while the reliability coefficients of `m` are .70 in one group and .80 in another.

Equal Numbers of Indicators and/or Reliability Coefficients:

If all variables in the model has the same number of indicators, `number_of_indicators` can be set to one single value.

Similarly, if all sets of indicators have the same population reliability in all groups, `reliability` can also be set to one single value.

Specify The Distributions of Exogenous Variables Or Error Terms Using 'x_fun'

By default, variables and error terms are generated from a multivariate normal distribution. If desired, users can supply the function used to generate an exogenous variable and error term by setting `x_fun` to a named list.

The names of the list are the variables for which a user function will be used to generate the data.

Each element of the list must also be a list. The first element of this list, can be unnamed, is the function to be used. If other arguments need to be supplied, they should be included as named elements of this list.

For example:

```
x_fun = list(x = list(power4mome::rexp_rs),
             w = list(power4mome::rbinary_rs,
                      p1 = .70)))
```

The variables `x` and `w` will be generated by user-supplied functions.

For `x`, the function is `power4mome::rexp_rs`. No additional argument when calling this function.

For `w`, the function is `power4mome::rbinary_rx`. The argument `p1 = .70` will be passed to this function when generating the values of `w`.

If a variable is an endogenous variable (e.g., being predicted by another variable in a model), then `x_fun` is used to generate its *error term*. Its implied population distribution may still be different from that generate by `x_fun` because the distribution also depends on the distribution of other variables predicting it.

These are requirements for the user-functions:

- They must return a numeric vector.
- They mush has an argument `n` for the number of values.
- The *population* mean and standard deviation of the generated values must be 0 and 1, respectively.

The package `power4mome` has helper functions for generating values from some common nonnormal distributions and then scaling them to have population mean and standard deviation equal to 0 and 1 (by default), respectively. These are some of them:

- `rbinary_rs()`.
- `rexp_rs()`.
- `rbeta_rs()`.
- `rlnorm_rs()`.
- `rpgnorm_rs()`.

To use `x_fun`, the variables must have zero covariances with other variables in the population. It is possible to generate nonnormal multivariate data but we believe this is rarely needed when estimating power *before* having the data.

Major Test-Related Arguments

The test function (test_fun):

The function set to `test_fun`, the *test function*, usually should work on the output of `lavaan::sem()`, `lmhelprs::many_lm()`, or `stats::lm()`, but can also be a function that works on the output of the function set to `fit_function` when calling `fit_model()` or `power4test()` (see `fit_model_args`).

The function has two default requirements.

First, it has an argument `fit`, to be set to the output of `lavaan::sem()` or another output stored in the element `extra$fit` of a replication in the `sim_all` object. (This requirement can be relaxed; see the section on `map_names`.)

That is, the function definition should be of this from: `FUN(fit, ...)`. This is the form of all `test_*` functions in `power4mome`.

If other arguments are to be passed to the test function, they can be set to `test_args` as a named list.

Second, the test function must returns an output that (a) can be processed by the results function (see below), or (b) is of the required format for the output of a results function (see the next section). If it already returns an output of the required format, then there is no need to set the results function.

The results function (results_fun):

The test results will be extracted from the output of `test_fun` by the function set to `results_fun`, the *results function*. If the `test_fun` already returns an output of the expected format (see below), then set `results_fun` to `NULL`, the default. The output of `test_fun` will be used for estimating power.

The function set to `results_fun` must accept the output of `test_fun`, as the first argument, and return a named list (which can be a data frame) or a named vector with some of the following elements:

- `est`: Optional. The estimate of a parameter, if applicable.
- `se`: Optional. The standard error of the estimate, if applicable.
- `cilo`: Optional. The lower limit of the confidence interval, if applicable.
- `cihi`: Optional. The upper limit of the confidence interval, if applicable.
- `sig`: Required. If `1`, the test is significant. If `0`, the test is not significant. If the test cannot be done for any reason, it should be `NA`.

The results can then be used to estimate the power or Type I error of the test.

For example, if the null hypothesis is false, then the proportion of significant, that is, the mean of the values of `sig` across replications, is the power.

Built-in test functions:

The package `power4mome` has some ready-to-use test functions:

- `test_indirect_effect()`
- `test_cond_indirect()`
- `test_cond_indirect_effects()`
- `test_moderation()`
- `test_index_of_mome()`
- `test_parameters()`

Please refer to their help pages for examples.

The argument map_names:

This argument is for developers using a test function that has a different name for the argument of the fit object ("fit", the default).

If test_fun is set to a function that works on an output of, say, lavaan::sem() but the argument name for the output is not fit, the mapping can be changed by map_names.

For example, lavaan::parameterEstimates() receives an output of lavaan::sem() and reports the test results of model parameters. However, the argument name for the lavaan output is object. To instruct do_test() to do the test correctly when setting test_fun to lavaan::parameterEstimates, add map_names = c(object = "fit"). The element fit in a replication will then set to the argument object of lavaan::parameterEstimates().

Examples

```
# Specify the model

model_simple_med <-
"
m ~ x
y ~ m + x
"

# Specify the population values

model_simple_med_es <-
"
m ~ x: m
y ~ m: l
y ~ x: n
"

# Set nrep to a large number in real analysis, such as 400
# Set `parallel` to TRUE for faster, usually much faster, analysis
# Set `progress` to TRUE to display the progress of the analysis

out <- power4test(nrep = 10,
                  model = model_simple_med,
                  pop_es = model_simple_med_es,
                  n = 100,
                  test_fun = test_parameters,
                  test_args = list(pars = "m~x"),
                  iseed = 1234,
                  parallel = FALSE,
                  progress = TRUE)

print(out,
      test_long = TRUE)

# Change the sample size

out1 <- power4test(out,
```

```
                    n = 200,
                    iseed = 1234,
                    parallel = FALSE,
                    progress = TRUE)

print(out1,
      test_long = TRUE)

# Add one more test

out2 <- power4test(out,
                    test_fun = test_parameters,
                    test_args = list(pars = "y~x"),
                    parallel = FALSE,
                    progress = TRUE)

print(out2,
      test_long = TRUE)
```

---

power4test_by_es *Power By Effect Sizes*

Description

Estimate power for a set of effect sizes (population values of a model parameter).

Usage

```
power4test_by_es(
  object,
  pop_es_name = NULL,
  pop_es_values = NULL,
  progress = TRUE,
  ...,
  by_seed = NULL,
  by_nrep = NULL,
  save_sim_all = TRUE
)

## S3 method for class 'power4test_by_es'
c(..., sort = TRUE, skip_checking_models = FALSE)

as.power4test_by_es(original_object, pop_es_name)

## S3 method for class 'power4test_by_es'
print(x, print_all = FALSE, digits = 3, ...)
```

Arguments

object	A power4test object, or a power4test_by_es object. If it is a power4test_by_es object, the first element, which is a power4test object, will be used as the value of this argument.
pop_es_name	The name of the parameter. See the help page of ptable_pop() on the names for the argument pop_es.
pop_es_values	A numeric vector of the population values of the parameter specified in pop_es_names.
progress	Logical. Whether progress of the simulation will be displayed.
...	For power4test_by_es(), they are arguments to be passed to power4test(). For c.power4test_by_es(), they are power4test_by_es() outputs to be combined together. For the print method of the output of power4test_by_es(), they are arguments to be passed to the print method of the output of power4test() (print.power4test()).
by_seed	If set to a number, it will be used to generate the seeds for each call to power4test(). If NULL, the default, then seeds will still be randomly generated but the results cannot be easily reproduced.
by_nrep	If set to a number, it will be used to generate the number of replications (nrep) for each call to power4test(). If set to a numeric vector of the same length as pop_es_values, then these are the nrep values for each of the calls, allowing for different numbers of replications for the population values. If NULL, the default, then the original nrep will be used. This argument is used by x_from_power() for efficiency, and is rarely used when calling this function directly.
save_sim_all	If FALSE, the dataset in the power4test objects are not saved, to reduce the size of the output. Default is TRUE.
sort	WHen combining the objects, whether they will be sorted by population values. Default is TRUE.
skip_checking_models	Whether the check of the data generation model will be checked. Default is TRUE. Should be set to FALSE only when users are certain the they are based on the same model, or when the model is not saved (e.g., save_sim_all set to FALSE when the objects were generated). This argument is used by x_from_power() for efficiency, and is rarely used when calling the c method directly.
original_object	The object to be converted to a power4test_by_es object.
x	The object to be printed.
print_all	If TRUE, all elements in x, that is, the results of all sample sizes examined, will be printed. If FALSE, then only those of the first value of the parameter will be printed.
digits	The numbers of digits displayed after the decimal.

Details

The function power4test_by_es() regenerates datasets for a set of effect sizes (population values of a model parmeter) and does the stored tests in each of them.

Optionally, it can also be run on a object with no stored tests. In this case, additional arguments must be set to instruct power4test() on the tests to be conducted.

It is usually used to examine the power over a sets of effect sizes (population values).

The c method of power4test_by_es objects is used to combine tests from different runs of power4test_by_es().

The function as.power4test_by_es() is used to convert a power4test object to a power4test_by_es object, if it is not already one. Useful when concatenating power4test objects with power4test_by_es objects.

Value

The function power4test_by_es() returns a power4test_by_es object, which is a list of power4test objects, one for each population value of the parameter.

The method c.power4test_by_es() returns a power4test_by_es object with all the elements (tests for different values of pop_es_values) combined.

The function as.power4test_by_es() returns a power4test_by_es object converted from the input object.

The print-method of power4test_by_es objects returns the object invisibly. It is called for its side-effect.

See Also

power4test()

Examples

```
# Specify the model

model_simple_med <-
"
m ~ x
y ~ m + x
"


# Specify the population values

model_simple_med_es <-
"
m ~ x: m
y ~ m: l
y ~ x: n
"

sim_only <- power4test(nrep = 2,
                       model = model_simple_med,
                       pop_es = model_simple_med_es,
                       n = 100,
                       R = 40,
                       ci_type = "boot",
                       fit_model_args = list(fit_function = "lm"),
```

```
                        do_the_test = FALSE,
                        iseed = 1234)

test_out <- power4test(object = sim_only,
                        test_fun = test_indirect_effect,
                        test_args = list(x = "x",
                                         m = "m",
                                         y = "y",
                                         boot_ci = TRUE,
                                         mc_ci = FALSE))

out <- power4test_by_es(test_out,
                             pop_es_name = "y ~ m",
                             pop_es_values = c(.10, .20))
out_reject <- rejection_rates(out)
out_reject
```

---

power4test_by_n *Power By Sample Sizes*

Description

Estimate power for a set of sample sizes.

Usage

```
power4test_by_n(
  object,
  n = NULL,
  progress = TRUE,
  ...,
  by_seed = NULL,
  by_nrep = NULL,
  save_sim_all = TRUE
)

## S3 method for class 'power4test_by_n'
c(
  ...,
  sort = TRUE,
  skip_checking_models = FALSE,
  tolerance_if_std_by_monte_carlo =
    getOption("power4mome.tolerance_if_std_by_monte_carlo", default = 0.01)
)

as.power4test_by_n(original_object)

## S3 method for class 'power4test_by_n'
print(x, print_all = FALSE, ...)
```

Arguments

object	A power4test object, or a power4test_by_n object. If it is a power4test_by_n object, the first element, which is a power4test object, will be used as the value of this argument.
n	A numeric vector of sample sizes for which the simulation will be conducted.
progress	Logical. Whether progress of the simulation will be displayed.
...	For power4test_by_n(), they are arguments to be passed to power4test(). For c.power4test_by_n(), they are power4test_by_n() outputs to be combined together. For the print method of the output of power4test_by_n(), they are arguments to be passed to the print method of the output of power4test() (print.power4test()).
by_seed	If set to a number, it will be used to generate the seeds for each call to power4test(). If NULL, the default, then seeds will still be randomly generated but the results cannot be easily reproduced.
by_nrep	If set to a number, it will be used to generate the number of replications (nrep) for each call to power4test(). If set to a numeric vector of the same length as n, then these are the nrep values for each of the calls, allowing for different numbers of replications for the sample sizes. If NULL, the default, then the original nrep will be used. This argument is used by x_from_power() for efficiency, and is rarely used when calling this function directly.
save_sim_all	If FALSE, the dataset in the power4test objects are not saved, to reduce the size of the output. Default is TRUE.
sort	When combining the objects, whether they will be sorted by sample sizes. Default is TRUE.
skip_checking_models	Whether the check of the data generation model will be checked. Default is TRUE. Should be set to FALSE only when users are certain the they are based on the same model, or when the model is not saved (e.g., save_sim_all set to FALSE when the objects were generated). This argument is used by x_from_power() for efficiency, and is rarely used when calling the c method directly.
tolerance_if_std_by_monte_carlo	The tolerance to be used by all.equal() when checking population values. If standardization is conducted using error variances estimated by Monte Carlo simulation, then they may not be exactly the same across replications. This is the tolerance used to compare population values, to allow for minor variation due to simulation.
original_object	The object to be converted to a power4test_by_n object.
x	The object to be printed.
print_all	If TRUE, all elements in x, that is, the results of all sample sizes examined, will be printed. If FALSE, then only those of the first sample size will be printed.

Details

The function power4test_by_n() regenerates datasets for a set of sample sizes and does the stored tests in each of them.

Optionally, it can also be run on a object with no stored tests. In this case, additional arguments must be set to instruct power4test() on the tests to be conducted.

It is usually used to examine the power over a set of sample sizes.

The c method of power4test_by_n objects is used to combine tests from different runs of power4test_by_n().

The function as.power4test_by_n() is used to convert a power4test object to a power4test_by_n object, if it is not already one. Useful when concatenating power4test objects with power4test_by_n objects.

Value

The function power4test_by_n() returns a power4test_by_n object, which is a list of power4test objects, one for each sample size.

The method c.power4test_by_n() returns a power4test_by_n object with all the elements (tests for different sample sizes) combined.

The function as.power4test_by_n() returns a power4test_by_n object converted from the input object.

The print-method of power4test_by_n objects returns the object invisibly. It is called for its side-effect.

See Also

power4test()

Examples

```
# Specify the model

model_simple_med <-
"
m ~ x
y ~ m + x
"


# Specify the population values

model_simple_med_es <-
"
m ~ x: m
y ~ m: l
y ~ x: n
"

sim_only <- power4test(nrep = 2,
                       model = model_simple_med,
                       pop_es = model_simple_med_es,
                       n = 100,
                       R = 40,
                       ci_type = "boot",
                       fit_model_args = list(fit_function = "lm"),
```

```
                        do_the_test = FALSE,
                        iseed = 1234)

test_out <- power4test(object = sim_only,
                        test_fun = test_indirect_effect,
                        test_args = list(x = "x",
                                         m = "m",
                                         y = "y",
                                         boot_ci = TRUE,
                                         mc_ci = FALSE))

out <- power4test_by_n(test_out,
                        n = c(100, 110, 120))
out_reject <- rejection_rates(out)
out_reject
```

`predict.power_curve` *Predict Method for a 'power_curve' Object*

Description

Compute the predicted values in a model fitted by `power_curve()`.

Usage

```
## S3 method for class 'power_curve'
predict(object, newdata, ...)
```

Arguments

`object`	A `power_curve` object.
`newdata`	A data frame with a column named `x`. It can also be a named list, with one element named `x` and is a vector of the values. If not supplied, values of `x` stored in `object` will be used.
`...`	Additional arguments. Passed to the corresponding `predict` method.

Details

The `predict` method of `power_curve` objects works in two modes.

If new data is not supplied (through `newdata`), it retrieves the stored results and calls the corresponding methods to compute the predicted values, which are the predicted rejection rates (power levels if the null hypothesis is false, e.g., the population effect size is equal to zero).

If new data is supplied, such as a named list with a vector of sample sizes, they will be used to compute the predicted rejection rates.

Value

It returns a numeric vector of the predicted rejection rates.

See Also

power_curve().

Examples

```
# Specify the population model

model_simple_med <-
"
m ~ x
y ~ m + x
"

# Specify the effect sizes (population parameter values)

model_simple_med_es <-
"
y ~ m: l
m ~ x: m
y ~ x: s
"

# Simulate datasets to check the model
# Set `parallel` to TRUE for faster, usually much faster, analysis
# Set `progress` to TRUE to display the progress of the analysis

sim_only <- power4test(nrep = 5,
                       model = model_simple_med,
                       pop_es = model_simple_med_es,
                       n = 50,
                       fit_model_args = list(fit_function = "lm"),
                       do_the_test = FALSE,
                       iseed = 1234,
                       parallel = FALSE,
                       progress = FALSE)

# By n: Do a test for different sample sizes

out1 <- power4test_by_n(sim_only,
                        nrep = 5,
                        test_fun = test_parameters,
                        test_args = list(par = "y~x"),
                        n = c(25, 100, 200, 1000),
                        by_seed = 1234,
                        parallel = FALSE,
                        progress = FALSE)

pout1 <- power_curve(out1)
```

```
pout1
predict(pout1,
        newdata = list(x = c(150, 250, 500)))
```

---

ptable_pop *Generate the Population Model*

Description

Generate the complete population model using the model syntax and user-specified effect sizes (population parameter values).

Usage

```
ptable_pop(
  model = NULL,
  pop_es = NULL,
  es1 = c(n = 0, nil = 0, s = 0.1, m = 0.3, l = 0.5, si = 0.141, mi = 0.361, li = 0.51,
    sm = 0.2, ml = 0.4),
  es2 = c(n = 0, nil = 0, s = 0.05, m = 0.1, l = 0.15, sm = 0.075, ml = 0.125),
  es_ind = c("si", "mi", "li"),
  standardized = TRUE,
  n_std = 1e+05,
  std_force_monte_carlo = FALSE,
  add_cov_for_moderation = TRUE
)

model_matrices_pop(x, ..., drop_list_single_group = TRUE)
```

Arguments

model	String. The model defined by lavaan model syntax. See 'Details'.
pop_es	It can be a data frame with these columns: lhs, op, rhs, and pop. The first three columns correspond to those in a lavaan parameter table. The column pop stores the population values. The column es stores the original labels, for reference. It can also be a named character vector (named list for multigroup models) or a multiline string, which are the preferred approaches. See the help page on how to specify this vector.
es1	Set the values for each label of the effect size (population value) for correlations and regression paths. Used only if pop_es is a named vector or a multiline string. See the help page on how to specify this argument.
es2	Set the values for each label of the effect size (population value) for product term. Used only if pop_es is a named vector or a multiline string. See the help page on how to specify this argument.

es_ind
: The names of labels denoting the effect size of an indirect effect. They will be used to determine the population values of the component paths along an indirect path.

standardized
: Logical. If TRUE, the default, variances and error variances are scaled to ensure the population variances of the endogenous variables are close to one, and hence the effect sizes (population values) are standardized effect sizes if the variances of the continuous exogenous variables are also equal to one.

n_std
: The sample size used to determine the error variances by simulation when std_force_monte_carlo is TRUE.

std_force_monte_carlo
: Logical. If FALSE, the default, standardization is done analytically if the model has no product terms, and by simulation if the model has product terms. That is, error variances required to ensure implied variances equal to one are determined by simulation. If TRUE, simulation will be used whether the model has product terms or not. Always fall back to simulation if analytical standardization failed.

add_cov_for_moderation
: Logical. If TRUE, the default, for a model in which one or product terms for moderation involve one or more mediator, covariances between their error terms and the product terms will be added automatically. If these covariances are not added, the model may not be invariant to linear transformation of some variables in the model.

x
: It can be a 'lavaan' model syntax, to be passed to ptable_pop(), or a parameter table with the column start set to the population values, such as the output of ptable_pop().

...
: If x is a model syntax, these are arguments to be passed to ptable_pop().

drop_list_single_group
: If TRUE and the number of groups is equal to one, the output will be a list of matrices of one group only. Default if TRUE.

Details

The function ptable_pop() generates a lavaan parameter table that can be used to generate data based on the population values of model parameters.

Value

The function ptable_pop() returns a lavaan parameter table of the model, with the column start set to the population values.

The function model_matrices_pop() returns a lavaan LISREL-style model matrices (like the output of lavaan::lavInspect() with what set to "free"), with matrix elements set to the population values. If x is the model syntax, it will be stored in the attributes model. If the model is a multigroup model with *k* groups (*k* greater than 1), then it returns a list of *k* lists of lavaan LISREL-style model matrices unless drop_list_single_group is TRUE.

The role of ptable_pop()

The function ptable_pop() is used by the all-in-one function power4test(). Users usually do not call this function directly, though developers can use this function to develop other functions for power analysis, or to build their own workflows to do the power analysis.

Specify the Population Model by 'model'

Single-Group Model:

For a single-group model, model should be a lavaan model syntax string of the *form* of the model. The population values of the model parameters are to be determined by pop_es.

If the model has latent factors, the syntax in model should specify only the *structural model* for the *latent factors*. There is no need to specify the measurement part. Other functions will generate the measurement part on top of this model.

For example, this is a simple mediation model:

```
"m ~ x
 y ~ m + x"
```

Whether m, x, and y denote observed variables or latent factors are determined by other functions, such as power4test().

Multigroup Model:

Because the model is the population model, equality constraints are irrelevant and the model syntax specifies only the *form* of the model. Therefore, model is specified as in the case of single group models.

Specify 'pop_es' Using Named Vectors

The argument pop_es is for specifying the population values of model parameters. This section describes how to do this using named vectors.

Single-Group Model:

If pop_es is specified by a named vector, it must follow the convention below.

- The names of the vectors are lavaan names for the selected parameters. For example, m ~ x denotes the path from x to m.
- Alternatively, the names can be either ".beta." or ".cov.". Use ".beta." to set the default values for all regression coefficients. Use ".cov." to set the default values for all correlations of exogenous variables (e.g., predictors).
- The names can also be of this form: ".ind.(<path>)", whether <path> denote path in the model. For example, ".ind.(x->m->y)" denotes the path from x through m to y. Alternatively, the lavaan symbol ~ can also be used: ".ind.(y~m~x)". This form is used to set the indirect effect (standardized, by default) along this path. The value for this name will override other settings.
- If using lavaan names, can specify more than one parameter using +. For example, y ~ m + x denotes the two paths from m and x to y.
- The value of each element can be the label for the effect size: n for nil, s for small, m for medium, and l for large. The value for each label is determined by es1 and es2. See the section on specifying these two arguments.

- The value of pop_es can also be set to a value, but it must be quoted as a string, such as "y ~ x" = ".31".

This is an example:

```
c(".beta." = "s",
  "m1 ~ x" = "-m",
  "m2 ~ m1" = "l",
  "y ~ x:w" = "s")
```

In this example,

- All regression coefficients are set to small (s) by default, unless specified otherwise.
- The path from x to m1 is set to medium and negative (-m).
- The path from m1 to m2 is set to large (l).
- The coefficient of the product term x:w when predicting y is set to small (s).

Indirect Effect:
When setting an indirect effect to a symbol (default: "si", "mi", "li", with "i" added to differentiate them from the labels for a direct path), the corresponding value is used to determine the population values of *all* component paths along the pathway. All the values are assumed to be equal. Therefore, ".ind.(x->m->y)" = ".20" is equivalent to setting m ~ x and y ~ m to the square root of .20, such that the corresponding indirect effect is equal to the designated value.
This behavior, though restricted, is for quick manipulation of the indirect effect. If different values along a pathway, set the value for each path directly.
Only nonnegative value is supported. Therefore, ".ind.(x->m->y)" = "-si" and ".ind.(x->m->y)" = "-.20" will throw an error.

Multigroup Model:

The argument pop_es also supports multigroup models.

For pop_es, instead of named vectors, named *list* of named vectors should be used.

- The names are the parameters, or keywords such as .beta. and .cov., like specifying the population values for a single group model.
- The elements are character vectors. If it has only one element (e.g., a single string), then it is the the population value for all groups. If it has more than one element (e.g., a vector of three strings), then they are the population values of the groups. For a model of *k* groups, each vector must have either *k* elements or one element.

This is an example:

```
list("m ~ x" = "m",
     "y ~ m" = c("s", "m", "l"))
```

In this model, the population value of the path m ~ x is medium (m) for all groups, while the population values for the path y ~ m are small (s), medium (m), and large (l), respectively.

Specify 'pop_es' Using a Multiline String

When setting the argument pop_es, instead of using a named vector or named list for pop_es, the population values of model parameters can also be specified using a multiline string, as illustrated below, to be parsed by pop_es_yaml().

Single-Group Model:

This is an example of the multiline string for a single-group model:

```
y ~ m: l
m ~ x: m
y ~ x: nil
```

The string must follow this format:

- Each line starts with `tag:`.
  - `tag` can be the name of a parameter, in `lavaan` model syntax format.
    - For example, `m ~ x` denotes the path from `x` to `m`.
  - A tag in `lavaan` model syntax can specify more than one parameter using `+`.
    - For example, `y ~ m + x` denotes the two paths from `m` and `x` to `y`.
  - Alternatively, the `tag` can be either `.beta.` or `.cov.`.
    - Use `.beta.` to set the default values for all regression coefficients.
    - Use `.cov.` to set the default values for all correlations of exogenous variables (e.g., predictors).
- After each tag is the value of the population value:
  -`nil` for nil (zero),
  - `s` for small,
  - `m` for medium, and
  - `l` for large.
  - `si`, `mi`, and `li` for small, medium, and large a standardized indirect effect, respectively.

  Note: `n` *cannot* be used in this mode.
  The value for each label is determined by `es1` and `es2` as described in `ptable_pop()`.
  - The value can also be set to a numeric value, such as `.30` or `-.30`.

This is another example:

```
.beta.: s
y ~ m: l
```

In this example, all regression coefficients are `small`, while the path from `m` to `y` is large.

Multigroup Model:

This is an example of the string for a multigroup model:

```
y ~ m: l
m ~ x:
  - nil
  - s
y ~ x: nil
```

The format is similar to that for a single-group model. If a parameter has the same value for all groups, then the line can be specified as in the case of a single-group model: `tag: value`.

If a parameter has different values across groups, then it must be in this format:

- A line starts with the tag, followed by two or more lines. Each line starts with a hyphen `-` and the value for a group.

For example:

```
m ~ x:
  - nil
  - s
```

This denotes that the model has two groups. The values of the path from `x` to `m` for the two groups are 0 (`nil`) and small (`s`), respectively.

Another equivalent way to specify the values are using `[]`, on the same line of a tag.

For example:

```
m ~ x: [nil, s]
```

The number of groups is inferred from the number of values for a parameter. Therefore, if a tag has more than one value, each tag must has the same number of value, or only one value.

The tag `.beta.` and `.cov.` can also be used for multigroup models.

Which Approach To Use:

Note that using named vectors or named lists is more reliable. However, using a multiline string is more user-friendly. If this method failed, please use named vectors or named list instead.

Technical Details:

The multiline string is parsed by `yaml::read_yaml()`. Therefore, the format requirement is actually that of YAML. Users knowledgeable of YAML can use other equivalent way to specify the string.

Set the Values for Effect Size Labels ('es1' and 'es2')

The vector `es1` is for correlations, regression coefficients, and indirect effect, and the vector `es2` is for for standardized moderation effect, the coefficients of a product term. These labels are to be used in interpreting the specification in `pop_es`.

The role of `model_matrices_pop()`

The function `model_matrices_pop()` generates models matrices with population values, used by `ptable_pop()`. Users usually do not call this function directly, though developers can use this build their own workflows to generate the data.

See Also

`power4test()`, and `pop_es_yaml()` on an alternative way to specify population values.

Examples

```
# Specify the model

model1 <-
"
m1 ~ x + c1
m2 ~ m1 + x2 + c1
y ~  m2 + m1 + x + w + x:w + c1
```

```
"

# Specify the population values

model1_es <- c("m1 ~ x" = "-m",
               "m2 ~ m1" = "s",
               "y ~ m2" = "1",
               "y ~ x" = "m",
               "y ~ w" = "s",
               "y ~ x:w" = "s",
               "x ~~ w" = "s")

ptable_final1 <- ptable_pop(model1,
                            pop_es = model1_es)
ptable_final1

# Use a multiline string, illustrated by a simpler model

model2 <-
"
m ~ x
y ~ m + x
"

model2_es_a <- c("m ~ x" = "s",
                 "y ~ m" = "m",
                 "y ~ x" = "nil")

model2_es_b <-
"
m ~ x: s
y ~ m: m
y ~ x: nil
"

ptable_model2_a <- ptable_pop(model2,
                              pop_es = model2_es_a)
ptable_model2_b <- ptable_pop(model2,
                              pop_es = model2_es_b)

ptable_model2_a
ptable_model2_b

identical(ptable_model2_a,
          ptable_model2_b)

# model_matrices_pop

model_matrices_pop(ptable_final1)

model_matrices_pop(model1,
                   pop_es = model1_es)
```

q_power_mediation *All-in-One Power Estimation For Mediation Models*

Description

All-in-one functions for estimating power or finding the region with target power for common mediation models.

Usage

```
q_power_mediation(
  model = NULL,
  pop_es = NULL,
  number_of_indicators = NULL,
  reliability = NULL,
  loading_difference = NULL,
  reference = NULL,
  test_fun = NULL,
  test_more_args = list(),
  target_power = 0.8,
  nrep = NULL,
  n = NULL,
  R = 1000,
  ci_type = c("mc", "boot"),
  seed = NULL,
  iseed = NULL,
  parallel = TRUE,
  progress = TRUE,
  simulation_progress = NULL,
  max_trials = NULL,
  algorithm = NULL,
  ...,
  mode = c("power", "region", "n")
)

## S3 method for class 'q_power_mediation'
print(x, mode = c("all", "region", "n", "power"), ...)

## S3 method for class 'q_power_mediation'
plot(x, ...)

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

q_power_mediation_simple(
  a = "m",
  b = "m",
```

```
  cp = "n",
  number_of_indicators = NULL,
  reliability = NULL,
  loading_difference = NULL,
  reference = NULL,
  test_more_args = list(),
  target_power = 0.8,
  nrep = NULL,
  n = NULL,
  R = 1000,
  ci_type = c("mc", "boot"),
  seed = NULL,
  iseed = NULL,
  parallel = TRUE,
  progress = TRUE,
  simulation_progress = NULL,
  max_trials = NULL,
  algorithm = NULL,
  ...,
  mode = c("power", "region", "n")
)

q_power_mediation_serial(
  ab = c("m", "m"),
  ab_other = "n",
  cp = "n",
  number_of_indicators = NULL,
  reliability = NULL,
  loading_difference = NULL,
  reference = NULL,
  test_more_args = list(),
  target_power = 0.8,
  nrep = NULL,
  n = NULL,
  R = 1000,
  ci_type = c("mc", "boot"),
  seed = NULL,
  iseed = NULL,
  parallel = TRUE,
  progress = TRUE,
  simulation_progress = NULL,
  max_trials = NULL,
  algorithm = NULL,
  ...,
  mode = c("power", "region", "n")
)

q_power_mediation_parallel(
```

```
  as = c("m", "m"),
  bs = c("m", "m"),
  cp = "n",
  number_of_indicators = NULL,
  reliability = NULL,
  loading_difference = NULL,
  reference = NULL,
  omnibus = c("all_sig", "at_least_one_sig", "at_least_k_sig"),
  at_least_k = 1,
  test_more_args = list(),
  target_power = 0.8,
  nrep = NULL,
  n = NULL,
  R = 1000,
  ci_type = c("mc", "boot"),
  seed = NULL,
  iseed = NULL,
  parallel = TRUE,
  progress = TRUE,
  simulation_progress = NULL,
  max_trials = NULL,
  algorithm = NULL,
  ...,
  mode = c("power", "region", "n")
)
```

Arguments

`model`	The `lavaan` model syntax of the population model, to be used by `ptable_pop()`. See 'Details' of on how to specify the model. Ignored if `ptable` is specified.
`pop_es`	The character vector or multiline string to specify population effect sizes (population values of parameters). See the help page on how to specify this argument. Ignored if `ptable` is specified.
`number_of_indicators`	A named vector to specify the number of indicators for each factors. See the help page on how to set this argument. Default is `NULL` and all variables in the model syntax are observed variables. See the help page on how to use this argument.
`reliability`	A named vector (for a single-group model) or a named list of named vectors (for a multigroup model) to set the reliability coefficient of each set of indicators. Default is `NULL`. See the help page on how to use this argument.
`loading_difference`	A named vector (for a single-group model) or a named list of named vectors (for a multigroup model) to set the difference in factor loadings between neighboring indicators of each set of indicators. Default is `NULL`, and all indicators of a factor have the same factor loadings. If specified, must be specified for all factors named in `reliability`, even for those with all loadings equal.
`reference`	A named vector (for a single-group model) or a named list of named vectors (for a multigroup model) to indicate which indicator will be the first indicator (and

so is the reference indicator, by default). Default is NULL, and for all factors, the indicator with the medium loading in a factor is the first indicator. Has no effect if loading difference is zero (and so all indicators have the same loadings). If specified, must be specified for all factors named in reliability, even for those with all loadings equal. Accepted values are "medium", "weakest", and "strongest".

test_fun — A function to do the test. See 'Details' of power4test() for the requirement of this function.

test_more_args — A named list of additional arguments to be passed to the test function (test_indirect_effect() for simple and serial mediation models, and test_k_indirect_effects() for parallel mediation models). Similar to test_args in power4test().

target_power — The target power, a value greater than 0 and less than one.

nrep — The number of replications to generate the simulated datasets. Default is NULL and will be determined internally.

n — The sample size for the first run of power4test(). Must be set to a sample size if mode is "power". For the other modes, if NULL, the default, it will be determined internally.

R — The number of replications to generate the Monte Carlo or bootstrapping estimates for each fit output. No Monte Carlo nor bootstrapping estimates will be generated if R is set to NULL.

ci_type — The type of simulation-based confidence intervals to use. Can be either "mc" for Monte Carlo method (the default) or "boot" for nonparametric bootstrapping method. Relevant for test functions that make use of estimates generate by gen_boot() or gen_mc(), such as test_indirect_effect().

seed — The seed for the random number generator. Used by n_region_from_power().

iseed — The seed for the random number generator. Used by power4test().

parallel — If TRUE, parallel processing will be used when calling other functions, if appropriate.

progress — If TRUE, the progress of each step will be displayed. Default is TRUE.

simulation_progress — Logical. Whether the progress in each call to power4test(), power4test_by_n(), or power4test_by_es() is shown. To be passed to the progress argument of these functions. If NULL, set automatically based on the algorithm used.

max_trials — The maximum number of trials in searching the value with the target power. Rounded up if not an integer. If NULL, set automatically based on the algorithm used.

algorithm — The algorithm to be used in mode "region" and "n". If NULL, then it will be determined internally based on the mode. (The default may be different from that of n_region_from_power() and n_from_power())

... — For q_power_mediation_*, these are optional arguments to be passed to power4test() and n_region_from_power(). For the print method, these are optional arguments to be passed to rejection_rates() as well as other print methods (see print.power4test() and print.n_region_from_power()). For the plot method, these are optional arguments to be passed to plot.n_region_from_power(). For the summary method, these are optional arguments to be passed to summary.n_region_from_power()

mode	For q_power_mediation_*, this is the mode of what to do. If "power", then only [power4tes gion", then the region of sample sizes with levels of power not significantly different f then all available outputs will be printed.
x	The object for the relevant methods.
object	For the summary method of q_power_mediation() outputs.
a	For a simple mediation model, this is the population effect size for the path from x to m .
b	For a simple mediation model, this is the population effect size for the path from m to y .
cp	For a simple mediation model, this is the population effect size for the direct path from c to y .
ab	For a serial mediation model, this is a numeric vector of the population effect sizes along the path x->m1->m2->...->y.
ab_other	Should be one single value. This is the population effect sizes of all other paths not along x->m1->m2->...->y, except for the direct path from x to y.
as	For a parallel mediation model, this is a numeric vector of the population effect sizes for the paths from x to the mediators: x->m1, x->m2, ... x->mp, for a parallel mediation model with p mediators.
bs	For a parallel mediation model, this is a numeric vector of the population effect sizes for the paths from the mediators to y: m1->y, m2->y, ... mp->y, for a parallel mediation model with p mediators.
omnibus	"all_sig", the default, then the test is declared significant if *all* paths are significant. If "at_least_one_sig", then only one row of test is stored, and the test is declared significant if at least one of the paths is significant. If "at_least_k_sig", then only one row of test is stored, and the test is declared significant if at least k of the paths is significant, k determined by the argument at_least_k.
at_least_k	The minimum number of paths required to be significant for the omnibus test to be considered significant. Used when omnibus is "at_least_k_sig".

Value

A named list of outputs. If mode is power, then a power4test object is set to the element power4test. If mode is region, then a n_region_from_power object is set of the element n_region_from_power. If mode is n_from_power, then a n_from_power object is set to the n_from_power element.

The print method of q_power_mediation returns x invisibly. Called for its side effect.

The plot-method of q_power_mediation returns NULL. It will plot either the output of n_region_from_power() (mode "region") or the output of n_from_power() (mode "n"). If no output from either of these functions is available, nothing wil be plotted.

The summary method for q_power_mediation objects returns the output of summary() for either the output of n_region_from_power() (mode "region") or the output of n_from_power() (mode "n"). Return NULL if no output from either of these functions is available.

All-in-One Functions for Common Mediation Models

These functions are wrappers that call `power4test()` and `n_region_from_power()` to (a) estimate the level of power for a mediation model, given the population effects and the sample size, and (b) find the region of sample sizes with the levels of power not significantly different from the target power.

They are convenient functions that set the argument values automatically for common mediation models before calling `power4test()` and `n_region_from_power()`. Please refer to the help pages of these two functions for the details on how the estimation and the search are conducted.

For some arguments not described in details here, please refer to the help pages of `power4test()` and `n_region_from_power()`,

Simple Mediation Model

The function `q_power_mediation_simple()` can be used for the power analysis of a simple mediation model with only one mediator.

This function will fit the following model:

```
"m ~ x
 y ~ m + x"
```

Serial Mediation Model

The function `q_power_mediation_serial()` can be used for the power analysis of a serial mediation model with only any number of mediators.

This is the model being fitted if the model has two mediators:

```
"m1 ~ x
 m2 ~ m1 + x
 y ~ m2 + m1 + x"
```

Parallel Mediation Model

The function `q_power_mediation_parallel()` can be used for the power analysis of a parallel mediation model with only any number of mediators.

This is the model being fitted if the model has two mediators:

```
"m1 ~ x
 m2 ~ x
 y ~ m2 + m1 + x"
```

An Arbitrary Mediation Model

The function `q_power_mediation()`, an advanced function, can be used for the power analysis of an arbitrary mediation model. The model and the population effect sizes are specified as in `power4test()`.

This is an example of a model with both parallel paths and serial paths:

```
model <-
 "
 m1 ~ x
 m21 ~ m1
 m22 ~ m1
 y ~ m21 + m22 + x
 "


pop_es <-
"
m1 ~ x: m
m21 ~ m1: m
m22 ~ m1: m
y ~ m21: m
y ~ m22: m
"
```

Knowledge of using `power4test()` is required to use this advanced function.

If this advanced function is used, users need to specify `test_fun` as when using `power4test()`, and need to set `test_args` correctly

See Also

See `power4test()` and `n_region_from_power()` for full details on how these functions work.

Examples

```
## Not run:

# An arbitrary mediation model

model <-
"
m1 ~ x
m21 ~ m1
m22 ~ m1
y ~ m21 + m22
"
pop_es <-
"
m1 ~ x: m
m21 ~ m1: m
m22 ~ m1: m
y ~ m21: m
y ~ m22: m
"

# NOTE: In real power analysis:
# - Set R to an appropriate value.
# - Remove nrep or set nrep to the desired value.
```

```
# - Remove parallel or set it to TRUE to enable parallel processing.
# - Remove progress or set it to TRUE to see the progress.

outa1 <- q_power_mediation(
    model = model,
    pop_es = pop_es,
    n = 100,
    R = 199,
    test_fun = test_k_indirect_effects,
    test_more_args = list(x = "x",
                          y = "y",
                          omnibus = "all"),
    seed = 1234,
    mode = "region",
    nrep = 20,
    parallel = FALSE,
    progress = FALSE
  )
outa1
summary(outa1)
plot(outa1)


## End(Not run)


# Simple mediation model

# NOTE: In real power analysis:
# - Set R to an appropriate value.
# - Remove nrep or set nrep to the desired value.
# - Remove parallel or set it to TRUE to enable parallel processing.
# - Remove progress or set it to TRUE to see the progress.

out <- q_power_mediation_simple(
    a = "m",
    b = "m",
    cp = "n",
    n = 50,
    R = 79,
    seed = 1234,
    nrep = 10,
    parallel = FALSE,
    progress = FALSE
  )
out

# If mode = "region" is added, can call the following
# summary(out)
# plot(out)


## Not run:
```

```
# Serial mediation model

# NOTE: In real power analysis:
# - Set R to an appropriate value.
# - Remove nrep or set nrep to the desired value.
# - Remove parallel or set it to TRUE to enable parallel processing.
# - Remove progress or set it to TRUE to see the progress.

outs <- q_power_mediation_serial(
    ab = c("s", "m", "l"),
    ab_others = "n",
    cp = "s",
    n = 50,
    R = 199,
    seed = 1234,
    mode = "region",
    nrep = 20,
    parallel = FALSE,
    progress = FALSE
  )
outs
summary(outs)
plot(outs)

## End(Not run)


## Not run:
# Parallel mediation model

# NOTE: In real power analysis:
# - Set R to an appropriate value.
# - Remove nrep or set nrep to the desired value.
# - Remove parallel or set it to TRUE to enable parallel processing.
# - Remove progress or set it to TRUE to see the progress.

outp <- q_power_mediation_parallel(
    as = c("s", "m"),
    bs = c("m", "s"),
    cp = "n",
    n = 100,
    R = 199,
    seed = 1234,
    mode = "region",
    nrep = 20,
    parallel = FALSE,
    progress = FALSE
  )
outp
summary(outp)
plot(outp)

## End(Not run)
```

rbeta_rs *Random Variable From a Beta Distribution*

Description

Generate random numbers from a beta distribution, rescaled to have user-specified population mean and standard deviation.

Usage

```
rbeta_rs(n = 10, shape1 = 0.5, shape2 = 0.5, pmean = 0, psd = 1)
```

Arguments

n	The number of random numbers to generate.
shape1	shape1 for stats::rbeta().
shape2	shape2 for stats::rbeta().
pmean	Population mean.
psd	Population standard deviation.

Details

First, specify the two parameters, shape1 and shape2, and the desired population mean and standard deviation. The random numbers, drawn from a beta distribution by stats::rbeta() will then be rescaled with the desired population mean and standard deviation.

Value

A vector of the generated random numbers.

Examples

```
set.seed(90870962)
x <- rbeta_rs(n = 5000,
              shape1 = .5,
              shape2 = .5,
              pmean = 3,
              psd = 1)
mean(x)
sd(x)
hist(x)
```

rbeta_rs2 *Random Variable From a Beta Distribution (User Range)*

Description

Generate random numbers from a beta distribution, rescaled to have user-specified population mean and standard deviation, and within a specific range.

Usage

```
rbeta_rs2(n = 10, bmean, bsd, blow = 0, bhigh = 1)
```

Arguments

n	The number of random numbers to generate.
bmean	The population mean.
bsd	The population standard deviation. If bsd is zero or negative, all random numbers will be equal to bmean.
blow	The lower bound of the target range.
bhigh	The upper bound of the target range.

Details

First, specify the two parameters, shape1 and shape2, and the desired population mean and standard deviation. The random numbers, drawn from a beta distribution by stats::rbeta() will then be rescaled to the desired population range.

Value

A vector of the generated random numbers.

Examples

```
set.seed(90870962)
x <- rbeta_rs2(n = 5000,
               bmean = .80,
               bsd = .10,
               blow = .00,
               bhigh = .95)
mean(x)
sd(x)
hist(x)
y <- rbeta_rs2(n = 5000,
               bmean = 4,
               bsd = 3,
               blow = -10,
               bhigh = 10)
mean(y)
```

```
sd(y)
hist(y)
```

`rbinary_rs` *Random Binary Variable*

Description

Generate random numbers from a distribution of 0 or 1, rescaled to have user-specified population mean and standard deviation.

Usage

```
rbinary_rs(n = 10, p1 = 0.5, pmean = 0, psd = 1)
```

Arguments

`n`	The number of random numbers to generate.
`p1`	The probability of being 1, before rescaling.
`pmean`	Population mean.
`psd`	Population standard deviation.

Details

First, specify probability of 1 (`p1`), and the desired population mean and standard deviation. The random numbers, drawn from a distribution of 0 (`1 - p1` probability) and 1 (`p1` probability), will then be rescaled with the desired population mean and standard.

Value

A vector of the generated random numbers.

Examples

```
set.seed(90870962)
x <- rbinary_rs(n = 5000,
                p1 = .5,
                pmean = 3,
                psd = 1)
mean(x)
sd(x)
hist(x)
```

`rejection_rates` *Rejection Rates*

Description

Get all rejection rates of all tests stored in a `power4test` object or other supported objects.

Usage

```
rejection_rates(object, ...)

## Default S3 method:
rejection_rates(object, ...)

## S3 method for class 'power4test'
rejection_rates(
  object,
  all_columns = FALSE,
  ci = TRUE,
  level = 0.95,
  se = FALSE,
  collapse = NULL,
  at_least_k = NULL,
  merge_all_tests = NULL,
  p_adjust_method = NULL,
  alpha = NULL,
  keep_nrep = FALSE,
  ...
)

## S3 method for class 'power4test_by_es'
rejection_rates(
  object,
  all_columns = FALSE,
  ci = TRUE,
  level = 0.95,
  se = FALSE,
  nrep_if_diff = TRUE,
  ...
)

## S3 method for class 'power4test_by_n'
rejection_rates(
  object,
  all_columns = FALSE,
  ci = TRUE,
  level = 0.95,
```

```
  se = FALSE,
  nrep_if_diff = TRUE,
  ...
)

## S3 method for class 'x_from_power'
rejection_rates(object, ...)

## S3 method for class 'n_region_from_power'
rejection_rates(object, ...)

## S3 method for class 'q_power_mediation'
rejection_rates(object, ...)

## S3 method for class 'rejection_rates_df'
print(x, digits = 3, annotation = TRUE, abbreviate_col_names = TRUE, ...)
```

Arguments

object	The object from which the rejection rates are to be computed, such as a power4test object, a power4test_by_n object, or a power4test_by_es object.
...	Optional arguments. For the print method, these arguments will be passed to the print method of data.frame objects print.data.frame(). For the rejection_rates method of x_from_power and n_region_from_power objects, these are arguments to be passed to the rejection_rates method for power4test_by_n and power4test_by_es objects. For the rejection_rates method for q_power_mediation objects (the output of q_power_mediation() and friends), they are optional arguments to be passed to the corresponding methods.
all_columns	If TRUE, all columns stored by a test will be extracted. Default is FALSE and only essential columns related to power will be printed.
ci	If TRUE, confidence intervals for the rejection rates (column reject or sig) will be computed. The method is determined by the option power4mome.ci_method. If NULL or "wilson", Wilson's (1927) method is used. If "norm", normal approximation is used.
level	The level of confidence for the confidence intervals, if ci is TRUE. Default is .95, denoting 95%.
se	If TRUE, standard errors for the rejection rates (column reject or sig) will be computed. Normal approximation is used to compute the standard errors.
collapse	Whether a single decision (significant vs. not significant) is made across all tests for a test that consists of several tests (e.g., the tests of several parameters). If "none", tests will be summarized individually. If "all_sig", then the set of tests is considered significant if all individual tests are significant. If "at_least_one_sig", then the set of tests is considered significant if at least one of the tests is significant. If "at_least_k_sig", then the set of tests is considered significant if at least k tests are significant, k set by the argument at_least_k. If NULL, will use the value stored in object (default is "none").

at_least_k
: Used by collapse, the number of tests required to be significant for the set of tests to be considered significant. If NULL, will use the value stored in object (default is 1).

merge_all_tests
: If TRUE, all the tests in each replication will be merged into one test. If NULL, will use the value stored in object (default is FALSE).

p_adjust_method
: The method to be passed to p.adjust() to adjust the *p*-values when testing the effects. Default is "none" and the *p*-values will not be adjusted. Ignored if some tests do not have *p*-values stored. NOTE: Use this only if all tests can be conducted using *p*-values. If NULL, will use the value stored in object (default is "none").

alpha
: The level of significance to use when using p_adjust_method. The significance results (the column sig) will be updated using the adjusted *p*-values. Used only if p_adjust_method is not "none". If NULL, will use the value stored in object (default is .05).

keep_nrep
: If TRUE, the column nrep will be kept.

nrep_if_diff
: If TRUE and the numbers of replications across rows are different, nrep will be included in the output. If FALSE, nrep will not be included unless all_columns is TRUE.

x
: The rejection_rates_df object to be printed.

digits
: The number of digits to be printed after the decimal.

annotation
: Logical. Whether additional notes will be printed.

abbreviate_col_names
: Logical. Whether some column names will be abbreviated.

Details

For a power4test object, rejection_rates loops over the tests stored in a power4test object and retrieves the rejection rate of each test.

The rejection_rates method for power4test_by_es objects is used to compute the rejection rates from a power4test_by_es object, with effect sizes added to the output.

The rejection_rates method for power4test_by_n objects is used to compute the rejection rates, with sample sizes added to the output.

The rejection_rates method for x_from_power objects is used to compute the rejection rates for stored trials. It supports the output of x_from_power() and its wrappers, such as n_from_power().

The rejection_rates method for n_region_from_power objects is used to compute the rejection rates for stored trials. It supports the output of n_region_from_power(). It is sufficient to retrieve the trials in searching for the upper bound because they also include the trials used in searching for the lower bound.

The rejection_rates method for q_power_mediation objects is used to compute the rejection rates for stored trials.

Value

The rejection_rates method returns a rejection_rates_df object, with a print method.

If the input (object) is a power4test object, the rejection_rates_df object is a data-frame like object with the number of rows equal to the number of tests. Note that some tests, such as the test by test_parameters(), conduct one test for each parameter. Each such test is counted as one test.

The data frame has at least these columns:

- test: The name of the test.
- label: The label for each test, or "Test" if a test only does one test (e.g., test_indirect_effect()).
- pvalid: The proportion of valid tests across all replications.
- reject: The rejection rate for each test. If the null hypothesis is false, then this is the power.

The rejection_rates method for power4test_by_es objects returns an object of the class rejection_rates_df_by_es, which is a subclass of rejection_rates_df. It is a data frame which is similar to the output of rejection_rates(), with two columns added for the effect size (pop_es_name and pop_es_values) for each test.

The rejection_rates method for power4test_by_n objects returns an object of the class rejection_rates_df_by_n, which is a subclass of rejection_rates_df. It is a data frame which is similar to the output of a power4test object, with a column n added for the sample size for each test.

The rejection_rates method for x_from_power objects retrieves the stored power4test_by_n or power4test_by_es object, and then runs rejection_rates on it and returns the result.

The rejection_rates method for n_region_from_power objects retrieves the stored power4test_by_n object from the above element (the search for the region with power significantly above the target power) and then runs rejection_rates on it and returns the result.

The rejection_rates method for q_power_mediation objects retrieves the trials from and then runs rejection_rates on them and returns the result. If mode is "n", then the stored output from n_from_power() is used. If mode is "region", then the stored output from n_region_from_power() is used. If mode is "power", then the stored output from power4test() is used.

The print method of a rejection_rates_df object returns the object invisibly. It is called for its side-effect.

References

Wilson, E. B. (1927). Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association, 22*(158), 209-212. doi:10.1080/01621459.1927.10502953

See Also

power4test(), power4test_by_n(), and power4test_by_es(), which are supported by this method.

Examples

```
# Specify the population model

model_simple_med <-
"
```

```
m ~ x
y ~ m + x
"

# Specify the effect sizes (population parameter values)

model_simple_med_es <-
"
y ~ m: l
m ~ x: m
y ~ x: n
"

# Generate some datasets to check the model

sim_only <- power4test(nrep = 4,
                       model = model_simple_med,
                       pop_es = model_simple_med_es,
                       n = 100,
                       R = 50,
                       ci_type = "boot",
                       fit_model_args = list(fit_function = "lm"),
                       do_the_test = FALSE,
                       iseed = 1234)

# Do the test 'test_indirect_effect' on each datasets

test_out <- power4test(object = sim_only,
                       test_fun = test_indirect_effect,
                       test_args = list(x = "x",
                                        m = "m",
                                        y = "y",
                                        boot_ci = TRUE,
                                        mc_ci = FALSE))

# Do the test 'test_parameters' on each datasets
# and add the results to 'test_out'

test_out <- power4test(object = test_out,
                       test_fun = test_parameters)

# Compute and print the rejection rates for stored tests

rejection_rates(test_out)

# See the help pages of power4test_by_n() and power4test_by_es()
# for other examples.
```

---

rexp_rs *Random Variable From an Exponential Distribution*

Description

Generate random numbers from an exponential distribution, rescaled to have user-specified population mean and standard deviation.

Usage

```
rexp_rs(n = 10, rate = 1, pmean = 0, psd = 1, rev = FALSE)
```

Arguments

n	The number of random numbers to generate.
rate	rate for stats::rexp().
pmean	Population mean.
psd	Population standard deviation.
rev	If TRUE, the distribution is revered to generate a negatively skewed distribution. Default is FALSE.

Details

First, specify the parameter, rate, and the desired population mean and standard deviation. The random numbers, drawn from an exponential distribution by stats::rexp(), will then be rescaled with the desired population mean and standard.

Value

A vector of the generated random numbers.

Examples

```
set.seed(90870962)
x <- rexp_rs(n = 5000,
             rate = 4,
             pmean = 3,
             psd = 1)
mean(x)
sd(x)
hist(x)
```

---

rlnorm_rs *Random Variable From a Lognormal Distribution*

Description

Generate random numbers from a lognormal distribution, rescaled to have user-specified population mean and standard deviation.

Usage

```
rlnorm_rs(n = 10, mui = 0, sigma = 1, rev = FALSE, pmean = 0, psd = 1)
```

Arguments

n	The number of random numbers to generate.
mui	The parameter mui to be used by stats::rlnorm().
sigma	The parameter sigma to be used by stats::rlnorm().
rev	If TRUE, the distribution is revered to generate a negatively skewed distribution. Default is FALSE.
pmean	Population mean.
psd	Population standard deviation.

Details

First, specify the parameter, mui and sigma, and the desired population mean and standard deviation. The random numbers, drawn from a lognormal distribution by stats::rlnorm(), will then be rescaled with the desired population mean and standard.

Value

A vector of the generated random numbers.

Examples

```
set.seed(90870962)
x <- rlnorm_rs(n = 5000,
               mui = 0,
               sigma = 1,
               pmean = 0,
               psd = 1)
mean(x)
sd(x)
hist(x)
```

---

rpgnorm_rs *Random Variable From a Generalized Normal Distribution*

Description

Generate random numbers from generalized normal distribution, rescaled to have user-specified population mean and standard deviation.

Usage

```
rpgnorm_rs(n = 10, p = 2, pmean = 0, psd = 1)
```

Arguments

n	The number of random numbers to generate.
p	The parameter of the distribution. Must be a positive non-zero number. Default is 2, resulting in a normal distribution. Higher than 2 results in negative excess kurtosis. Lower than 2 results in positive excess kurtosis.
pmean	Population mean.
psd	Population standard deviation.

Details

First, specify the parameter p and the desired population mean and standard deviation. The random numbers, drawn from a generalized normal distribution by `pgnorm::rpgnorm()`, will then be rescaled with the desired population mean and standard.

Value

A vector of the generated random numbers.

Examples

```
set.seed(90870962)
x <- rpgnorm_rs(n = 5000,
                p = 2,
                pmean = 0,
                psd = 1)
mean(x)
sd(x)
hist(x)
x_kurt <- function(p) {gamma(5/p)*gamma(1/p)/(gamma(3/p)^2) - 3}

p <- 6
x <- rpgnorm_rs(n = 50000,
                p = p,
                pmean = 0,
                psd = 1)
mean(x)
sd(x)
x_kurt(p)
qqnorm(x); qqline(x)

p <- 1
x <- rpgnorm_rs(n = 50000,
                p = p,
                pmean = 0,
                psd = 1)
mean(x)
sd(x)
x_kurt(p)
qqnorm(x); qqline(x)
```

rt_rs *Random Variable From a t Distribution*

Description

Generate random numbers from a *t* distribution, rescaled to have user-specified population mean and standard deviation.

Usage

```
rt_rs(n = 10, df = 5, pmean = 0, psd = 1)
```

Arguments

n	The number of random numbers to generate.
df	df for stats::rt().
pmean	Population mean.
psd	Population standard deviation.

Details

First, specify the parameter df and the desired population mean and standard deviation. The random numbers, drawn from the generalized normal distribution by stats::rt(), will then be rescaled with the desired population mean and standard.

Value

A vector of the generated random numbers.

Examples

```
set.seed(90870962)
x <- rt_rs(n = 5000,
           df = 5,
           pmean = 3,
           psd = 1)
mean(x)
sd(x)
hist(x)
```

runif_rs *Random Variable From a Uniform Distribution*

Description

Generate random numbers from a uniform distribution, with user-specified population mean and standard deviation.

Usage

```
runif_rs(n = 10, min = 0, max = 1, pmean = 0, psd = 1)
```

Arguments

n	The number of random numbers to generate.
min	min for runif.
max	max for runif.
pmean	Population mean.
psd	Population standard deviation.

Details

First, the user specifies the parameters, min and max, and the desired population mean and standard deviation. Then the random numbers will be generated and rescaled with the desired population mean and standard.

Value

A vector of the generated random numbers.

Examples

```
set.seed(90870962)
x <- runif_rs(n = 5000,
              min = 2,
              max = 4,
              pmean = 3,
              psd = 1)
mean(x)
sd(x)
hist(x)
```

scale_scores *Process Data by Computing Scale Scores*

Description

For the process_data argument. To compute scale scores from indicators and replace the indicators scores by computed scale scores.

Usage

```
scale_scores(data, method = c("mean", "sum"))
```

Arguments

data	A data frame with the indicator scores. It must has an attribute number_of_indicators. The same argument used by power4test(). This attribute is used to identify the factor names and their indicators.
method	The method to be used to compute the scale scores. Can be "mean" or "sum". The default na.rm = FALSE will be used. Therefore, data must not have missing data.

Details

This function is to be used in the process_data argument of power4test().

It retrieves the attribute "number_of_indicators", stored by power4test(), to identify factors with indicators, computes the scale scores based on method, and replace the indicators by the scale scores.

All subsequent steps, such as the test functions, will see only the scale scores, or original scores if a variable has no indicator. The model will also be fitted on the scale scores, not on the indicators.

It can be used to estimate power for analyzing the scale scores, taking into account the measurement error due to imperfect reliability.

Value

It returns a data frame with the scale scores computed.

See Also

power4test()

Examples

```
# Specify the model

mod <-
"
m ~ x
```

```
y ~ m + x
"

# Specify the population values

mod_es <-
"
y ~ m: l
m ~ x: m
y ~ x: n
"

# Specify the numbers of indicators and reliability coefficients

k <- c(y = 3,
       m = 4,
       x = 5)
rel <- c(y = .70,
         m = .70,
         x = .70)

# Simulate the data

out <- power4test(
         nrep = 2,
         model = mod,
         pop_es = mod_es,
         n = 200,
         number_of_indicators = k,
         reliability = rel,
         process_data = list(fun = "scale_scores"),
         test_fun = test_parameters,
         test_args = list(op = "~"),
         parallel = FALSE,
         iseed = 1234)

dat <- pool_sim_data(out)
head(dat)
```

`sim_data` *Simulate Datasets Based on a Model*

Description

Get a model matrix and effect size specification and simulate a number of datasets, along with other information.

The function

Usage

```
sim_data(
  nrep = 10,
  ptable = NULL,
  model = NULL,
  pop_es = NULL,
  ...,
  n = 100,
  iseed = NULL,
  number_of_indicators = NULL,
  reliability = NULL,
  loading_difference = NULL,
  reference = NULL,
  x_fun = list(),
  e_fun = list(),
  process_data = NULL,
  parallel = FALSE,
  progress = FALSE,
  ncores = max(1, parallel::detectCores(logical = FALSE) - 1),
  n_ratio = 1,
  cl = NULL
)

## S3 method for class 'sim_data'
print(
  x,
  digits = 3,
  digits_descriptive = 2,
  data_long = TRUE,
  fit_to_all_args = list(),
  est_type = "standardized",
  variances = NULL,
  pure_x = TRUE,
  pure_y = TRUE,
  ...
)

pool_sim_data(object, as_list = FALSE)
```

Arguments

nrep	The number of replications to generate the simulated datasets. Default is 10.
ptable	The output of ptable_pop(), which is a ptable_pop object, representing the population model. If NULL, the default, ptable_pop() will be called to generate the ptable_pop object, using arguments such as model and pop_es.
model	The lavaan model syntax of the population model. Ignored if ptable is specified. See ptable_pop on how to specify this argument.

pop_es
: The character to specify population effect sizes. See ptable_pop on how to specify this argument. Ignored if ptable is specified.

...
: For sim_data, parameters to be passed to ptable_pop(). For print.sim_data(), these arguments are ignored.

n
: The sample size for each dataset. Default is 100.

iseed
: The seed for the random number generator. Default is NULL and the seed is not changed.

number_of_indicators
: A named vector to specify the number of indicators for each factors. See the help page on how to set this argument. Default is NULL and all variables in the model syntax are observed variables. See the help page on how to use this argument.

reliability
: A named vector (for a single-group model) or a named list of named vectors (for a multigroup model) to set the reliability coefficient of each set of indicators. Default is NULL. See the help page on how to use this argument.

loading_difference
: A named vector (for a single-group model) or a named list of named vectors (for a multigroup model) to set the difference in factor loadings between neighboring indicators of each set of indicators. Default is NULL, and all indicators of a factor have the same factor loadings. If specified, must be specified for all factors named in reliability, even for those with all loadings equal.

reference
: A named vector (for a single-group model) or a named list of named vectors (for a multigroup model) to indicate which indicator will be the first indicator (and so is the reference indicator, by default). Default is NULL, and for all factors, the indicator with the medium loading in a factor is the first indicator. Has no effect if loading difference is zero (and so all indicators have the same loadings). If specified, must be specified for all factors named in reliability, even for those with all loadings equal. Accepted values are "medium", "weakest", and "strongest".

x_fun
: The function(s) used to generate the exogenous variables or error terms. If not supplied, or set to list(), the default, the variables are generated from a multivariate normal distribution. See the help page on how to use this argument.

e_fun
: The function(s) used to generate the error terms of indicators, if any. If not supplied, or set to list(), the default, the error terms of indicators are generated from a multivariate normal distribution. Specify in the same way as x_fun. Refer to the help page on x_fun on how to use this argument.

process_data
: If not NULL, it must be a named list with these elements: fun (required), the function to further process the simulated data, such as generating missing data using functions such as mice::ampute(); args (optional), a named list of arguments to be passed to fun, except the one for the source data; sim_data_name (optional) the name of the argument to receive the simulated data (e.g., "data" for mice::ampute()), default to "data" if it is not set; processed_data_name (optional), the name of the data frame after being processed by fun, such as the data frame with missing data in the output of fun (e.g., "amp" for mice::ampute()), if omitted, the output of fun should be the data frame with missing data.

parallel
: If TRUE, parallel processing will be used to simulate the datasets. Default is FALSE.

progress — If TRUE, the progress of data simulation will be displayed. Default is 'FALSE.

ncores — The number of CPU cores to use if parallel processing is used.

n_ratio — If the model is a multigroup model, and n is a single number, this should be a numeric vector used to determine the sample size for each group. For example, for a two-group model, if n is 100 and n_ratio is c(1, 0.5), then the sample sizes for the two groups are 100 and 50, respectively. If equal to 1, then all groups have the same sample size.

cl — A cluster, such as one created by parallel::makeCluster(). If NULL, a cluster will be created, but will be stopped on exit. If set to an existing cluster, it will not be stopped when the function exits; users need to stop it manually.

x — The sim_data object to be printed.

digits — The numbers of digits displayed after the decimal.

digits_descriptive — The number of digits displayed after the decimal for the descriptive statistics table.

data_long — If TRUE, detailed information will be printed.

fit_to_all_args — A named list of arguments to be passed to lavaan::sem() when the model is fitted to a sample combined from all samples stored.

est_type — The type of estimates to be printed. Can be a character vector of one to two elements. If only "standardized", then the standardized estimates are printed. If only "unstandardized", then the unstandardized estimates are printed. If a vector like c("standardized", "unstandardized"), then both unstandardized and standardized estimates are printed.

variances — Logical. Whether variances and error variances are printed. Default depends on est_type. If "unstandardized" is in est_type, then default is TRUE If only "standardized" is in est_type, then default is FALSE.

pure_x, pure_y — When Logical. When printing indirect effects, whether only "pure" x-variables (variables not predicted by another other variables) and/or "pure" y-variables (variables that do not predict any other variables other than indicators) will be included in enumerating the paths.

object — Either a sim_data object or a power4test object. It extracts the simulated data and return them, combined to one single data frame or, if as_list is TRUE, as a list of data frames.

as_list — Logical. If TRUE, the simulated datasets is returned as one single data frame. If FALSE, they are returned as a list of data frames.

Details

The function sim_data() generates a list of datasets based on a population model.

Value

The function sim_out() returns a list of the class sim_data, with length nrep. Each element is a sim_data_i object, with the following major elements:

- ptable: A lavaan parameter table of the model, with population values set in the column start. (It is the output of the function ptable_pop().)
- mm_out: The population model represented by model matrices as in lavaan. (It is the output of the function model_matrices_pop().)
- mm_lm_out: A list of regression model formula, one for each endogenous variable. (It is the output of the internal function mm_lm().)
- mm_lm_dat_out: A simulated dataset generated from the population model. (It is the output of the internal function mm_lm_data()).
- model_original: The original model syntax (i.e., the argument model).
- model_final: A modified model syntax if the model is a latent variable model. Indicators are added to the syntax.
- fit0: The output of lavaan::sem() with ptable as the model and do.fit set to FALSE. Used for the easy retrieval of information about the model.

The print method of sim_data returns x invisibly. It is called for its side effect.

The function pool_sim_data() returns either one data frame or a list of data frames, depending on the argument as_list

The role of sim_data()

The function sim_data() is used by the all-in-one function power4test(). Users usually do not call this function directly, though developers can use this function to develop other functions for power analysis, or to build their own workflows to do the power analysis.

Workflow

The function sim_data() does two tasks:

- Determine the actual population model with population values based on:
  - A model syntax for the observed variables (for a path model) or the latent factors (for a latent variable model).
  - A textual specification of the effect sizes of parameters.
  - The number of indicators for each latent factor if the model is a latent variable model.
  - The reliability of each latent factor as measured by the indicators if the model is a latent factor model.
- Generate *nrep* simulated datasets from the population model.

The simulated datasets can then be used to fit a model, test parameters, and estimate power.

The output is usually used by fit_model() to fit a target model, by default the population model, to each of the dataset.

Set 'number_of_indicators' and 'reliability'

The arguments number_of_indicators and reliability are used to specify the number of indicators (e.g., items) for each factor, and the population reliability coefficient of each factor, if the variables in the model syntax are latent variables.

Optionally, loading_difference can be used to generate indicators with unequal standardized factor loadings, and reference can be used to specify the indicator with the medium, strongest, or weakest standardized factor loading as the first indicator, which is used as the reference indicator in lavaan.

Single-Group Model:

If a variable in the model is to be replaced by indicators in the generated data, set number_of_indicators to a named numeric vector. The names are the variables of variables with indicators, as appeared in the model syntax. The value of each name is the number of indicators.

The argument reliability should then be set a named numeric vector (or list, see the section on multigroup models) to specify the population reliability coefficient ("omega") of each set of indicators. The population standardized factor loadings are then computed to ensure that the population reliability coefficient is of the target value.

These are examples for a single group model:

```
number of indicator = c(m = 3, x = 4, y = 5)
```

The numbers of indicators for m, x, and y are 3, 4, and 5, respectively.

```
reliability = c(m = .90, x = .80, y = .70)
```

The population reliability coefficients of m, x, and y are .90, .80, and .70, respectively.

Multigroup Models:

Multigroup models are supported. The number of groups is inferred from pop_es (see the help page of ptable_pop()), or directly from ptable.

For a multigroup model, the number of indicators for each variable must be the same across groups.

However, the population reliability coefficients can be different across groups. For a multigroup model of *k* groups, with one or more population reliability coefficients differ across groups, the argument reliability should be set to a named list. The names are the variables to which the population reliability coefficients are to be set. The element for each name is either a single value for the common reliability coefficient, or a numeric vector of the reliability coefficient of each group.

This is an example of reliability for a model with 2 groups:

```
reliability = list(x = .80, m = c(.70, .80))
```

The reliability coefficients of x are .80 in all groups, while the reliability coefficients of m are .70 in one group and .80 in another.

Equal Numbers of Indicators and/or Reliability Coefficients:

If all variables in the model has the same number of indicators, number_of_indicators can be set to one single value.

Similarly, if all sets of indicators have the same population reliability in all groups, reliability can also be set to one single value.

Specify The Distributions of Exogenous Variables Or Error Terms Using 'x_fun'

By default, variables and error terms are generated from a multivariate normal distribution. If desired, users can supply the function used to generate an exogenous variable and error term by setting `x_fun` to a named list.

The names of the list are the variables for which a user function will be used to generate the data.

Each element of the list must also be a list. The first element of this list, can be unnamed, is the function to be used. If other arguments need to be supplied, they should be included as named elements of this list.

For example:

```
x_fun = list(x = list(power4mome::rexp_rs),
             w = list(power4mome::rbinary_rs,
                      p1 = .70)))
```

The variables `x` and `w` will be generated by user-supplied functions.

For `x`, the function is `power4mome::rexp_rs`. No additional argument when calling this function.

For `w`, the function is `power4mome::rbinary_rx`. The argument `p1 = .70` will be passed to this function when generating the values of `w`.

If a variable is an endogenous variable (e.g., being predicted by another variable in a model), then `x_fun` is used to generate its *error term*. Its implied population distribution may still be different from that generate by `x_fun` because the distribution also depends on the distribution of other variables predicting it.

These are requirements for the user-functions:

- They must return a numeric vector.
- They mush has an argument `n` for the number of values.
- The *population* mean and standard deviation of the generated values must be 0 and 1, respectively.

The package `power4mome` has helper functions for generating values from some common nonnormal distributions and then scaling them to have population mean and standard deviation equal to 0 and 1 (by default), respectively. These are some of them:

- `rbinary_rs()`.
- `rexp_rs()`.
- `rbeta_rs()`.
- `rlnorm_rs()`.
- `rpgnorm_rs()`.

To use `x_fun`, the variables must have zero covariances with other variables in the population. It is possible to generate nonnormal multivariate data but we believe this is rarely needed when estimating power *before* having the data.

Specify the Population Model by 'model'

Single-Group Model:

For a single-group model, `model` should be a `lavaan` model syntax string of the *form* of the model. The population values of the model parameters are to be determined by `pop_es`.

If the model has latent factors, the syntax in `model` should specify only the *structural model* for the *latent factors*. There is no need to specify the measurement part. Other functions will generate the measurement part on top of this model.

For example, this is a simple mediation model:

```
"m ~ x
 y ~ m + x"
```

Whether `m`, `x`, and `y` denote observed variables or latent factors are determined by other functions, such as `power4test()`.

Multigroup Model:

Because the model is the population model, equality constraints are irrelevant and the model syntax specifies only the *form* of the model. Therefore, `model` is specified as in the case of single group models.

Specify 'pop_es' Using Named Vectors

The argument `pop_es` is for specifying the population values of model parameters. This section describes how to do this using named vectors.

Single-Group Model:

If `pop_es` is specified by a named vector, it must follow the convention below.

- The names of the vectors are `lavaan` names for the selected parameters. For example, `m ~ x` denotes the path from `x` to `m`.
- Alternatively, the names can be either `".beta."` or `".cov."`. Use `".beta."` to set the default values for all regression coefficients. Use `".cov."` to set the default values for all correlations of exogenous variables (e.g., predictors).
- The names can also be of this form: `".ind.(<path>)"`, whether `<path>` denote path in the model. For example, `".ind.(x->m->y)"` denotes the path from `x` through `m` to `y`. Alternatively, the `lavaan` symbol `~` can also be used: `".ind.(y~m~x)"`. This form is used to set the indirect effect (standardized, by default) along this path. The value for this name will override other settings.
- If using `lavaan` names, can specify more than one parameter using `+`. For example, `y ~ m + x` denotes the two paths from `m` and `x` to `y`.
- The value of each element can be the label for the effect size: `n` for nil, `s` for small, `m` for medium, and `l` for large. The value for each label is determined by `es1` and `es2`. See the section on specifying these two arguments.
- The value of `pop_es` can also be set to a value, but it must be quoted as a string, such as `"y ~ x" = ".31"`.

This is an example:

```
c(".beta." = "s",
  "m1 ~ x" = "-m",
  "m2 ~ m1" = "l",
  "y ~ x:w" = "s")
```

In this example,

- All regression coefficients are set to small (`s`) by default, unless specified otherwise.
- The path from `x` to `m1` is set to medium and negative (`-m`).
- The path from `m1` to `m2` is set to large (`l`).
- The coefficient of the product term `x:w` when predicting `y` is set to small (`s`).

Indirect Effect:
When setting an indirect effect to a symbol (default: `"si"`, `"mi"`, `"li"`, with `"i"` added to differentiate them from the labels for a direct path), the corresponding value is used to determine the population values of *all* component paths along the pathway. All the values are assumed to be equal. Therefore, `".ind.(x->m->y)" = ".20"` is equivalent to setting `m ~ x` and `y ~ m` to the square root of .20, such that the corresponding indirect effect is equal to the designated value.
This behavior, though restricted, is for quick manipulation of the indirect effect. If different values along a pathway, set the value for each path directly.
Only nonnegative value is supported. Therefore, `".ind.(x->m->y)" = "-si"` and `".ind.(x->m->y)" = "-.20"` will throw an error.

Multigroup Model:

The argument `pop_es` also supports multigroup models.

For `pop_es`, instead of named vectors, named *list* of named vectors should be used.

- The names are the parameters, or keywords such as `.beta.` and `.cov.`, like specifying the population values for a single group model.
- The elements are character vectors. If it has only one element (e.g., a single string), then it is the the population value for all groups. If it has more than one element (e.g., a vector of three strings), then they are the population values of the groups. For a model of *k* groups, each vector must have either *k* elements or one element.

This is an example:

```
list("m ~ x" = "m",
     "y ~ m" = c("s", "m", "l"))
```

In this model, the population value of the path `m ~ x` is medium (`m`) for all groups, while the population values for the path `y ~ m` are small (`s`), medium (`m`), and large (`l`), respectively.

Specify 'pop_es' Using a Multiline String

When setting the argument `pop_es`, instead of using a named vector or named list for `pop_es`, the population values of model parameters can also be specified using a multiline string, as illustrated below, to be parsed by `pop_es_yaml()`.

Single-Group Model:

This is an example of the multiline string for a single-group model:

```
y ~ m: l
m ~ x: m
y ~ x: nil
```

The string must follow this format:

- Each line starts with `tag:`.
  - `tag` can be the name of a parameter, in `lavaan` model syntax format.
    - For example, `m ~ x` denotes the path from `x` to `m`.
  - A tag in `lavaan` model syntax can specify more than one parameter using `+`.
    - For example, `y ~ m + x` denotes the two paths from `m` and `x` to `y`.
  - Alternatively, the `tag` can be either `.beta.` or `.cov.`.
    - Use `.beta.` to set the default values for all regression coefficients.
    - Use `.cov.` to set the default values for all correlations of exogenous variables (e.g., predictors).
- After each tag is the value of the population value:
  -`nil` for nil (zero),
  - `s` for small,
  - `m` for medium, and
  - `l` for large.
  - `si`, `mi`, and `li` for small, medium, and large a standardized indirect effect, respectively.

  Note: `n` *cannot* be used in this mode.
  The value for each label is determined by `es1` and `es2` as described in `ptable_pop()`.
  - The value can also be set to a numeric value, such as `.30` or `-.30`.

This is another example:

```
.beta.: s
y ~ m: l
```

In this example, all regression coefficients are `small`, while the path from `m` to `y` is large.

Multigroup Model:

This is an example of the string for a multigroup model:

```
y ~ m: l
m ~ x:
  - nil
  - s
y ~ x: nil
```

The format is similar to that for a single-group model. If a parameter has the same value for all groups, then the line can be specified as in the case of a single-group model: `tag: value`.

If a parameter has different values across groups, then it must be in this format:

- A line starts with the tag, followed by two or more lines. Each line starts with a hyphen `-` and the value for a group.

For example:

```
m ~ x:
  - nil
  - s
```

This denotes that the model has two groups. The values of the path from `x` to `m` for the two groups are 0 (`nil`) and small (`s`), respectively.

Another equivalent way to specify the values are using `[]`, on the same line of a tag.

For example:

```
m ~ x: [nil, s]
```

The number of groups is inferred from the number of values for a parameter. Therefore, if a tag has more than one value, each tag must has the same number of value, or only one value.

The tag `.beta.` and `.cov.` can also be used for multigroup models.

Which Approach To Use:

Note that using named vectors or named lists is more reliable. However, using a multiline string is more user-friendly. If this method failed, please use named vectors or named list instead.

Technical Details:

The multiline string is parsed by `yaml::read_yaml()`. Therefore, the format requirement is actually that of YAML. Users knowledgeable of YAML can use other equivalent way to specify the string.

Set the Values for Effect Size Labels ('es1' and 'es2')

The vector `es1` is for correlations, regression coefficients, and indirect effect, and the vector `es2` is for for standardized moderation effect, the coefficients of a product term. These labels are to be used in interpreting the specification in `pop_es`.

See Also

`power4test()`

Examples

```
# Specify the model

mod <-
"m ~ x
 y ~ m + x"

# Specify the population values

es <-
"
y ~ m: m
m ~ x: m
y ~ x: n
"
```

```
# Generate the simulated datasets

data_all <- sim_data(nrep = 5,
                     model = mod,
                     pop_es = es,
                     n = 100,
                     iseed = 1234)

data_all
```

`sim_out` *Create a 'sim_out' Object*

Description

Combine the outputs of `sim_data()`, `fit_model()`, and optionally `gen_mc()` and/or `gen_boot()` to one single object.

Usage

```
sim_out(data_all, ...)

## S3 method for class 'sim_out'
print(x, digits = 3, digits_descriptive = 2, fit_to_all_args = list(), ...)
```

Arguments

`data_all`	The output of `sim_data()`.
`...`	Named arguments of objects to be added to each replication under the element `extra`. For example, if set to `fit = fit_all`, where `fit_all` is the output of `fit_model()`, then `data_all[[1]]$extra$fit` will be set to the first output in `fit_all`.
`x`	The `sim_out` object to be printed.
`digits`	The numbers of digits displayed after the decimal.
`digits_descriptive`	The number of digits displayed after the decimal for the descriptive statistics table.
`fit_to_all_args`	A named list of arguments to be passed to `lavaan::sem()` when the model is fitted to a sample combined from all samples stored.

Details

It merges into one object the output of sim_data(), which is a list of nrep simulated datasets, fit_model(), which is a list of the lavaan::sem() output for the nrep datasets, and optionally the output of gen_mc() or gen_boot(), which is a list of the R sets of Monte Carlo or bootstrap estimates based on the results of fit_model(). The list has nrep elements, each element with the data, the model fit results, and optionally the Monte Carlo estimates matched.

This object can then be used for testing effects of interests, which are further processed to estimate the power of this test.

The function sim_out() is used by the all-in-one function power4test(). Users usually do not call this function directly, though developers can use this function to develop other functions for power analysis, or to build their own workflows to do the power analysis.

Value

The function sim_out() returns a sim_out object, which is a list of length equal to the length of data_all. Each element of the list is a sim_data object with the element extra added to it. Other named elements will be added under this name. For example. the output of fit_model() for this replication can be added to fit, under extra. See the description of the argument ... for details.

The print method of sim_out returns x invisibly. Called for its side effect.

See Also

power4test()

Examples

```
# Specify the model

mod <-
"m ~ x
 y ~ m + x"

# Specify the population values

es <-
"
y ~ m: m
m ~ x: m
y ~ x: n
"

# Generate the simulated datasets

dats <- sim_data(nrep = 5,
                 model = mod,
                 pop_es = es,
                 n = 100,
                 iseed = 1234)

# Fit the population model to each dataset
```

```
fits <- fit_model(dats)

# Combine the results to one object

sim_out_all <- sim_out(data_all = dats,
                       fit = fits)
sim_out_all

# Verify that the elements of fits are set to extra$fit

library(lavaan)
parameterEstimates(fits[[1]])
parameterEstimates(sim_out_all[[1]]$extra$fit)
parameterEstimates(fits[[2]])
parameterEstimates(sim_out_all[[2]]$extra$fit)
```

---

summarize_tests *Summarize Test Results*

Description

Extract and summarize test results.

Usage

```
summarize_tests(
  object,
  collapse = c("none", "all_sig", "at_least_one_sig", "at_least_k_sig"),
  at_least_k = 1,
  merge_all_tests = FALSE,
  p_adjust_method = "none",
  alpha = 0.05
)

## S3 method for class 'test_summary_list'
print(x, digits = 3, ...)

## S3 method for class 'test_summary'
print(x, digits = 2, ...)

## S3 method for class 'test_out_list'
print(x, digits = 3, test_long = TRUE, ...)
```

Arguments

object	A power4test object or the element test_all in a power4test object.

collapse — Whether a single decision (significant vs. not significant) is made across all tests for a test that consists of several tests (e.g., the tests of several parameters). If "none", tests will be summarized individually. If "all_sig", then the set of tests is considered significant if all individual tests are significant. If "at_least_one_sig", then the set of tests is considered significant if at least one of the tests is significant. If "at_least_k_sig", then the set of tests is considered significant if at least k tests are significant, k set by the argument at_least_k.

at_least_k — Used by collapse, the number of tests required to be significant for the set of tests to be considered significant.

merge_all_tests — If TRUE, all the tests in each replication will be merged into one test.

p_adjust_method — The method to be passed to p.adjust() to adjust the *p*-values when testing the effects. Default is "none" and the *p*-values will not be adjusted. The unadjusted *p*-values will be stored in the column pvalue_org. Ignored if some tests do not have *p*-values stored. NOTE: Use this only if all tests can be conducted using *p*-values.

alpha — The level of significance to use when using p_adjust_method. The significance results (the column sig) will be updated using the adjusted *p*-values. Used only if p_adjust_method is not "none".

x — The object to be printed.

digits — The numbers of digits after the decimal when printing numeric results.

... — Optional arguments. Not used.

test_long — If TRUE, a detailed report will be printed.

Details

The function summarize_tests() is used to extract information from each test stored in a power4test object.

The method print.test_out_list() is used to print the content of a list of test stored in a power4test object, with the option to print just the names of tests.

Value

The function summarize_tests() returns a list of the class test_summary_list. Each element contains a summary of a test stored. The elements are of the class test_summary, with these elements:

- test_attributes: The stored information of a test, for printing.
- nrep: The number of datasets (replications).
- mean: The means of numeric information. For significance tests, these are the rejection rates.
- nvalid: The number of non-NA replications used to compute each mean.

The print methods returns x invisibly. They are called for their side effects.

The role of `summarize_tests()` and related functions

The function `summarize_tests()` and related print methods are used by the all-in-one function `power4test()` and its `summary` method. Users usually do not call them directly, though developers can use this function to develop other functions for power analysis, or to build their own workflows to do the power analysis.

See Also

`power4test()`

Examples

```
# Specify the model

mod <-
"
m ~ x
y ~ m + x
"

# Specify the population values

es <-
"
y ~ m: l
m ~ x: m
y ~ x: n
"

# Simulated datasets

sim_only <- power4test(nrep = 2,
                       model = mod,
                       pop_es = es,
                       n = 100,
                       do_the_test = FALSE,
                       iseed = 1234)

# Test the parameters in each dataset

test_out <- power4test(object = sim_only,
                       test_fun = test_parameters)

# Print the summary

summarize_tests(test_out)
```

summary.x_from_power *Summarize 'x_from_power' Results*

Description

The summary method of the output of x_from_power().

Usage

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

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

## S3 method for class 'summary.x_from_power'
print(x, digits = 3, ...)

## S3 method for class 'summary.n_region_from_power'
print(x, digits = 3, ...)
```

Arguments

object	An x_from_power-class object, such as the output of x_from_power(), or an object of the class n_region_from_power, such as the output of n_region_from_power().
...	Additional arguments. Not used for now.
x	The output of summary.x_from_power(), the summary method of an x_from_power object, which is the output of x_from_power(), or the output of summary.n_region_from_power(), the summary method of an n_region_from_power object (the output of n_region_from_power()).
digits	The number of digits after the decimal when printing the results.

Details

The summary method simply prepares the results of x_from_power() to be printed in details.

Value

The summary method for x_from_power objects returns an object of the class summary.x_from_power, which is simply the output of x_from_power(), with a print method dedicated for detailed summary. Please refer to x_from_power() for the contents.

The print-method of summary.x_from_power objects returns the object x invisibly. It is called for its side effect.

The print-method of summary.n_region_from_power objects returns the object x invisibly. It is called for its side effect.

See Also

x_from_power(), n_region_from_power()

Examples

```
# Specify the population model

mod <-
"
m ~ x
y ~ m + x
"

# Specify the population values

mod_es <-
"
m ~ x: m
y ~ m: l
y ~ x: n
"

# Generate the datasets

sim_only <- power4test(nrep = 5,
                       model = mod,
                       pop_es = mod_es,
                       n = 100,
                       do_the_test = FALSE,
                       iseed = 2345)

# Do a test

test_out <- power4test(object = sim_only,
                       test_fun = test_parameters,
                       test_args = list(pars = "m~x"))

# Determine the sample size with a power of .80 (default)

power_vs_n <- x_from_power(test_out,
                           x = "n",
                           progress = TRUE,
                           target_power = .80,
                           final_nrep = 5,
                           max_trials = 1,
                           seed = 1234)
summary(power_vs_n)
```

`test_cond_indirect` *Test a Conditional Indirect Effect*

Description

Test a conditional indirect effect for a `power4test` object.

Usage

```
test_cond_indirect(
  fit = fit,
  x = NULL,
  m = NULL,
  y = NULL,
  wvalues = NULL,
  mc_ci = TRUE,
  mc_out = NULL,
  boot_ci = FALSE,
  boot_out = NULL,
  check_post_check = TRUE,
  test_method = NULL,
  ...,
  fit_name = "fit",
  get_map_names = FALSE,
  get_test_name = FALSE
)
```

Arguments

`fit`	The fit object, to be passed to `manymome::cond_indirect()`.
`x`	The name of the `x`-variable, the predictor.
`m`	A character vector of the name(s) of mediator(s). The path moves from the first mediator in the vector to the last mediator in the vector. Can be `NULL` and the path is a direct path without mediator.
`y`	The name of the `y`-variable, the outcome variable.
`wvalues`	A numeric vector of named elements. The names are the variable names of the moderators, and the values are the values to which the moderators will be set to. Default is `NULL`.
`mc_ci`	Logical. If `TRUE`, the default, Monte Carlo confidence intervals will be formed. This argument and `boot_ci` cannot be both `TRUE`.
`mc_out`	The pre-generated Monte Carlo estimates generated by manymome::do_mc, stored in a `power4test` object. Users should not set this argument and should let `power4test()` to set it automatically.
`boot_ci`	Logical. If `TRUE`, the default, nonparametric bootstrap confidence intervals will be formed. This argument and `mc_ci` cannot be both `TRUE`.

boot_out
: The pre-generated bootstrap estimates generated by manymome::do_boot, stored in a power4test object. Users should not set this argument and should let power4test() to set it automatically.

check_post_check
: Logical. If TRUE, the default, and the model is fitted by lavaan, the test will be conducted only if the model passes the post.check conducted by lavaan::lavInspect() (with what = "post.check").

test_method
: The method to do the test. If "ci", then the confidence interval (e.g., bootstrap confidence interval) will be used to do the test. If "pvalue", then asymmetric *p*-value by Asparouhov & Muthén (2021) will be used to do the test, and the confidence interval will not be computed. If NULL, its value will be set to "pvalue" if the number of simulated/bootstrap samples (R) is a value that is supported by the method by Boos and Zhang (2000), and set to "ci" otherwise.

...
: Additional arguments to be passed to manymome::cond_indirect().

fit_name
: The name of the model fit object to be extracted. Default is "fit". Used only when more than one model is fitted in each replication. This should be the name of the model on which the test is to be conducted.

get_map_names
: Logical. Used by power4test() to determine how to extract stored information and assign them to this function. Users should not use this argument.

get_test_name
: Logical. Used by power4test() to get the default name of this test. Users should not use this argument.

Details

This function is to be used in power4test() for testing a conditional indirect effect, by setting it to the test_fun argument.

It uses manymome::cond_indirect() to do the test. It can be used on models fitted by lavaan::sem() or fitted by a sequence of calls to stats::lm(), although only nonparametric bootstrap confidence interval is supported for models fitted by regression using stats::lm().

It can also be used to test a conditional effect on a direct path with no mediator. Just omit m when calling the function.

Value

In its normal usage, it returns a named numeric vector with the following elements:

- est: The mean of the estimated indirect effect across datasets.
- cilo and cihi: The means of the lower and upper limits of the confidence interval (95% by default), respectively.
- sig: Whether a test by confidence interval is significant (1) or not significant (0).

See Also

power4test()

Examples

```
# Specify the model

model_simple_mod <-
"
m ~ x + w + x:w
y ~ m + x
"

# Specify the population values

model_simple_mod_es <-
"
y ~ m: l
y ~ x: n
m ~ x: m
m ~ w: n
m ~ x:w: l
"

# Simulate the data

sim_only <- power4test(nrep = 5,
                       model = model_simple_mod,
                       pop_es = model_simple_mod_es,
                       n = 100,
                       R = 100,
                       do_the_test = FALSE,
                       iseed = 1234)

# Do the test in each replication

test_ind <- power4test(object = sim_only,
                       test_fun = test_cond_indirect,
                       test_args = list(x = "x",
                                        m = "m",
                                        y = "y",
                                        wvalues = c(w = 1),
                                        mc_ci = TRUE))
print(test_ind,
      test_long = TRUE)
```

`test_cond_indirect_effects` *Test Several Conditional Indirect Effects*

Description

Test several conditional indirect effects for a `power4test` object.

Usage

```
test_cond_indirect_effects(
  fit = fit,
  x = NULL,
  m = NULL,
  y = NULL,
  wlevels = NULL,
  mc_ci = TRUE,
  mc_out = NULL,
  boot_ci = FALSE,
  boot_out = NULL,
  check_post_check = TRUE,
  test_method = NULL,
  compare_groups = FALSE,
  ...,
  fit_name = "fit",
  get_map_names = FALSE,
  get_test_name = FALSE
)
```

Arguments

fit	The fit object, to be passed to manymome::cond_indirect_effects().
x	The name of the x-variable, the predictor.
m	A character vector of the name(s) of mediator(s). The path moves from the first mediator in the vector to the last mediator in the vector. Can be NULL and the path is a direct path without mediator.
y	The name of the y-variable, the outcome variable.
wlevels	The output of manymome::merge_mod_levels(), or the moderator(s) to be passed to manymome::mod_levels_list(). If all the moderators can be represented by one variable, that is, each moderator is (a) a numeric variable, (b) a dichotomous categorical variable, or (c) a factor or string variable used in stats::lm() in fit, then it is a vector of the names of the moderators as appeared in the data frame. If at least one of the moderators is a categorical variable represented by more than one variable, such as user-created dummy variables used in lavaan::sem(), then it must be a list of the names of the moderators, with such moderators represented by a vector of names. For example: list("w1", c("gpgp2", "gpgp3"), the first moderator w1 and the second moderator a three-category variable represented by gpgp2 and gpgp3. See the help page of manymome::cond_indirect_effects() for further details.
mc_ci	Logical. If TRUE, the default, Monte Carlo confidence intervals will be formed. This argument and boot_ci cannot be both TRUE.
mc_out	The pre-generated Monte Carlo estimates generated by manymome::do_mc, stored in a power4test object. Users should not set this argument and should let power4test() to set it automatically.
boot_ci	Logical. If TRUE, the default, nonparametric bootstrap confidence intervals will be formed. This argument and mc_ci cannot be both TRUE.

boot_out
: The pre-generated bootstrap estimates generated by manymome::do_boot, stored in a power4test object. Users should not set this argument and should let power4test() to set it automatically.

check_post_check
: Logical. If TRUE, the default, and the model is fitted by lavaan, the test will be conducted only if the model passes the post.check conducted by lavaan::lavInspect() (with what = "post.check").

test_method
: The method to do the test. If "ci", then the confidence interval (e.g., bootstrap confidence interval) will be used to do the test. If "pvalue", then asymmetric *p*-value by Asparouhov & Muthén (2021) will be used to do the test, and the confidence interval will not be computed. If NULL, its value will be set to "pvalue" if the number of simulated/bootstrap samples (R) is a value that is supported by the method by Boos and Zhang (2000), and set to "ci" otherwise.

compare_groups
: If the model is a multigroup model, compute and test group differences for all pairwise combinations of the groups. Ignored if the model is a single-group model.

...
: Additional arguments to be passed to manymome::cond_indirect_effects().

fit_name
: The name of the model fit object to be extracted. Default is "fit". Used only when more than one model is fitted in each replication. This should be the name of the model on which the test is to be conducted.

get_map_names
: Logical. Used by power4test() to determine how to extract stored information and assign them to this function. Users should not use this argument.

get_test_name
: Logical. Used by power4test() to get the default name of this test. Users should not use this argument.

Details

This function is to be used in power4test() for testing several conditional indirect effects, by setting it to the test_fun argument.

It uses manymome::cond_indirect_effects() to do the test. It can be used on models fitted by lavaan::sem() or fitted by a sequence of calls to stats::lm(), although only nonparametric bootstrap confidence interval is supported for models fitted by regression using stats::lm().

It can also be used to test conditional effects on a direct path with no mediator. Just omit m when calling the function.

Value

In its normal usage, it returns the output returned by manymome::cond_indirect_effects(), with the following modifications:

- est: The estimated conditional indirect effects.
- cilo and cihi: The lower and upper limits of the confidence interval (95% by default), respectively.
- sig: Whether a test by confidence interval is significant (1) or not significant (0).
- test_label: A column of labels generated to label the conditional effects.

See Also

power4test()

Examples

```
# Specify the model

model_simple_mod <-
"
m ~ x + w + x:w
y ~ m + x
"

# Specify the population values

model_simple_mod_es <-
"
y ~ m: l
y ~ x: n
m ~ x: m
m ~ w: n
m ~ x:w: l
"

# Simulate the data

# Set nrep to a larger value in real analysis, such as 400
sim_only <- power4test(nrep = 5,
                       model = model_simple_mod,
                       pop_es = model_simple_mod_es,
                       n = 100,
                       R = 100,
                       do_the_test = FALSE,
                       iseed = 1234)

# Do the tests in each replication

test_out <- power4test(object = sim_only,
                       test_fun = test_cond_indirect_effects,
                       test_args = list(x = "x",
                                        m = "m",
                                        y = "y",
                                        wlevels = c("w"),
                                        mc_ci = TRUE))
print(test_out,
      test_long = TRUE)
```

---

test_group_equal *Test Group Constraints*

Description

Test the model fit change when one or more between-group constraints are imposed.

Usage

```
test_group_equal(
  fit = fit,
  group.equal = NULL,
  group.partial = NULL,
  check_post_check = TRUE,
  ...,
  fit_name = "fit",
  get_map_names = FALSE,
  get_test_name = FALSE
)
```

Arguments

`fit`	The fit object. Must be the output of `lavaan::lavaan()` or its wrappers, such as `lavaan::sem()` and `lavaan::cfa()`. The model must be a multigroup model.
`group.equal`	The same argument used by `lavaan`. A character vector with one or more of these values: `"regressions"`, `"loadings"`, `"lv.covariances"`, `"lv.variances"`, `"intercepts"`, `"means"`, `"thresholds"`, `"residual.covariances"`, `"composite.weights"`, and `"residuals"`.
`group.partial`	The same argument used by `lavaan`. The parameters that should be free across groups. Used with `group.equal` to exclude some parameters from those requested to be equal across groups by `group.equal`.
`check_post_check`	Logical. If `TRUE`, the default, and the model is fitted by `lavaan`, the test will be conducted only if the model passes the `post.check` conducted by `lavaan::lavInspect()` (with `what = "post.check"`).
`...`	Optional arguments to be passed to `lavaan::lavTestLRT()`.
`fit_name`	The name of the model fit object to be extracted. Default is `"fit"`. Used only when more than one model is fitted in each replication. This should be the name of the model on which the test is to be conducted.
`get_map_names`	Logical. Used by `power4test()` to determine how to extract stored information and assign them to this function. Users should not use this argument.
`get_test_name`	Logical. Used by `power4test()` to get the default name of this test. Users should not use this argument.

Details

This function is to be used in `power4test()` for testing the difference in model fit when one or more between-group constraints are imposed , by setting it to the `test_fun` argument.

Value

In its normal usage, it returns a one-row data frame with the following columns:

- `est`: The chi-square difference.
- `cilo` and `cihi`: `NA`. Not used.
- `sig`: Whether the chi-square difference test is significant
- `test_label`: The constraints imposted.

See Also

`power4test()`

Examples

```
# Specify the model

mod <-
"
m ~ x
y ~ m + x
"

# Specify the population values

mod_es <-
"
y ~ m: l
m ~ x:
  - nil
  - s
y ~ x: nil
"

# Simulate the data

sim_only <- power4test(nrep = 2,
                       model = mod,
                       pop_es = mod_es,
                       n = 100,
                       iseed = 1234)

# Do the tests in each replication

test_out <- power4test(object = sim_only,
                       test_fun = test_group_equal,
                       test_args = list(group.equal = "regressions"))

print(test_out,
      test_long = TRUE)
```

`test_index_of_mome` *Test a Moderated Mediation Effect*

Description

Test a moderated mediation effect for a `power4test` object.

Usage

```
test_index_of_mome(
  fit = fit,
  x = NULL,
  m = NULL,
  y = NULL,
  w = NULL,
  mc_ci = TRUE,
  mc_out = NULL,
  boot_ci = FALSE,
  boot_out = NULL,
  check_post_check = TRUE,
  test_method = NULL,
  ...,
  fit_name = "fit",
  get_map_names = FALSE,
  get_test_name = FALSE
)
```

Arguments

`fit`	The fit object, to be passed to `manymome::index_of_mome()`.
`x`	The name of the `x`-variable, the predictor.
`m`	A character vector of the name(s) of mediator(s). The path moves from the first mediator in the vector to the last mediator in the vector. Can be `NULL` and the path is a direct path without mediator.
`y`	The name of the `y`-variable, the outcome variable.
`w`	The name of the moderator.
`mc_ci`	Logical. If `TRUE`, the default, Monte Carlo confidence intervals will be formed. This argument and `boot_ci` cannot be both `TRUE`.
`mc_out`	The pre-generated Monte Carlo estimates generated by manymome::do_mc, stored in a `power4test` object. Users should not set this argument and should let `power4test()` to set it automatically.
`boot_ci`	Logical. If `TRUE`, the default, nonparametric bootstrap confidence intervals will be formed. This argument and `mc_ci` cannot be both `TRUE`.

`boot_out` The pre-generated bootstrap estimates generated by manymome::do_boot, stored in a `power4test` object. Users should not set this argument and should let `power4test()` to set it automatically.

`check_post_check`
Logical. If `TRUE`, the default, and the model is fitted by `lavaan`, the test will be conducted only if the model passes the `post.check` conducted by `lavaan::lavInspect()` (with `what = "post.check"`).

`test_method` The method to do the test. If `"ci"`, then the confidence interval (e.g., bootstrap confidence interval) will be used to do the test. If `"pvalue"`, then asymmetric *p*-value by Asparouhov & Muthén (2021) will be used to do the test, and the confidence interval will not be computed. If `NULL`, its value will be set to `"pvalue"` if the number of simulated/bootstrap samples (`R`) is a value that is supported by the method by Boos and Zhang (2000), and set to `"ci"` otherwise.

`...` Additional arguments to be passed to `manymome::index_of_mome()`.

`fit_name` The name of the model fit object to be extracted. Default is `"fit"`. Used only when more than one model is fitted in each replication. This should be the name of the model on which the test is to be conducted.

`get_map_names` Logical. Used by `power4test()` to determine how to extract stored information and assign them to this function. Users should not use this argument.

`get_test_name` Logical. Used by `power4test()` to get the default name of this test. Users should not use this argument.

Details

This function is to be used in `power4test()` for testing a moderated mediation effect, by setting it to the `test_fun` argument.

It uses `manymome::index_of_mome()` to do the test. It can be used on models fitted by `lavaan::sem()` or fitted by a sequence of calls to `stats::lm()`, although only nonparametric bootstrap confidence interval is supported for models fitted by regression using `stats::lm()`.

Value

In its normal usage, it returns a named numeric vector with the following elements:

- `est`: The mean of the estimated indirect effect across datasets.
- `cilo` and `cihi`: The means of the lower and upper limits of the confidence interval (95% by default), respectively.
- `sig`: Whether a test by confidence interval is significant (`1`) or not significant (`0`).

See Also

`power4test()`

Examples

```
# Specify the model

mod <-
"
m ~ x + w + x:w
y ~ m
"

# Specify the population values

mod_es <-
"
m ~ x: n
y ~ x: m
m ~ w: l
m ~ x:w: l
"

# Simulate the data

sim_only <- power4test(nrep = 2,
                       model = mod,
                       pop_es = mod_es,
                       n = 100,
                       R = 100,
                       do_the_test = FALSE,
                       iseed = 1234)

# Do the test in each replication

test_out <- power4test(object = sim_only,
                       test_fun = test_index_of_mome,
                       test_args = list(x = "x",
                                        m = "m",
                                        y = "y",
                                        w = "w",
                                        mc_ci = TRUE))

print(test_out,
      test_long = TRUE)
```

---

test_indirect_effect *Test an Indirect Effect*

Description

Test an indirect effect for a power4test object.

Usage

```
test_indirect_effect(
  fit = fit,
  x = NULL,
  m = NULL,
  y = NULL,
  mc_ci = TRUE,
  mc_out = NULL,
  boot_ci = FALSE,
  boot_out = NULL,
  check_post_check = TRUE,
  test_method = NULL,
  ...,
  fit_name = "fit",
  get_map_names = FALSE,
  get_test_name = FALSE
)
```

Arguments

fit	The fit object, to be passed to manymome::indirect_effect().
x	The name of the x-variable, the predictor.
m	A character vector of the name(s) of mediator(s). The path moves from the first mediator in the vector to the last mediator in the vector. Can be NULL and the path is a direct path without mediator.
y	The name of the y-variable, the outcome variable.
mc_ci	Logical. If TRUE, the default, Monte Carlo confidence intervals will be formed. This argument and boot_ci cannot be both TRUE.
mc_out	The pre-generated Monte Carlo estimates generated by manymome::do_mc, stored in a power4test object. Users should not set this argument and should let power4test() to set it automatically.
boot_ci	Logical. If TRUE, the default, nonparametric bootstrap confidence intervals will be formed. This argument and mc_ci cannot be both TRUE.
boot_out	The pre-generated bootstrap estimates generated by manymome::do_boot, stored in a power4test object. Users should not set this argument and should let power4test() to set it automatically.
check_post_check	Logical. If TRUE, the default, and the model is fitted by lavaan, the test will be conducted only if the model passes the post.check conducted by lavaan::lavInspect() (with what = "post.check").
test_method	The method to do the test. If "ci", then the confidence interval (e.g., bootstrap confidence interval) will be used to do the test. If "pvalue", then asymmetric *p*-value by Asparouhov & Muthén (2021) will be used to do the test, and the confidence interval will not be computed. If NULL, its value will be set to "pvalue" if the number of simulated/bootstrap samples (R) is a value that is supported by the method by Boos and Zhang (2000), and set to "ci" otherwise.

`...`	Additional arguments to be passed to `manymome::indirect_effect()`.
`fit_name`	The name of the model fit object to be extracted. Default is `"fit"`. Used only when more than one model is fitted in each replication. This should be the name of the model on which the test is to be conducted.
`get_map_names`	Logical. Used by `power4test()` to determine how to extract stored information and assign them to this function. Users should not use this argument.
`get_test_name`	Logical. Used by `power4test()` to get the default name of this test. Users should not use this argument.

Details

This function is to be used in `power4test()` for testing an indirect effect, by setting it to the `test_fun` argument.

It uses `manymome::indirect_effect()` to do the test. It can be used on models fitted by `lavaan::sem()` or fitted by a sequence of calls to `stats::lm()`, although only nonparametric bootstrap confidence interval is supported for models fitted by regression using `stats::lm()`.

Value

In its normal usage, it returns a named numeric vector with the following elements:

- `est`: The mean of the estimated indirect effect across datasets.
- `cilo` and `cihi`: The means of the lower and upper limits of the confidence interval (95% by default), respectively.
- `sig`: Whether a test by confidence interval is significant (`1`) or not significant (`0`).

References

Asparouhov, A., & Muthén, B. (2021). Bootstrap p-value computation. Retrieved from https://www.statmodel.com/download Bootstrap%20-%20Pvalue.pdf

See Also

`power4test()`

Examples

```
# Specify the model

model_simple_med <-
"
m ~ x
y ~ m + x
"


# Specify the population values

model_simple_med_es <-
"
```

```
y ~ m: l
m ~ x: m
y ~ x: n
"

# Simulate the data

sim_only <- power4test(nrep = 5,
                       model = model_simple_med,
                       pop_es = model_simple_med_es,
                       n = 100,
                       R = 100,
                       do_the_test = FALSE,
                       iseed = 1234)

# Do the test in each replication

test_ind <- power4test(object = sim_only,
                       test_fun = test_indirect_effect,
                       test_args = list(x = "x",
                                        m = "m",
                                        y = "y",
                                        mc_ci = TRUE))
print(test_ind,
      test_long = TRUE)
```

`test_k_indirect_effects` *Test Several Indirect Effects*

Description

Test several indirect effects for a `power4test` object.

Usage

```
test_k_indirect_effects(
  fit = fit,
  x = NULL,
  m = NULL,
  y = NULL,
  mc_ci = TRUE,
  mc_out = NULL,
  boot_ci = FALSE,
  boot_out = NULL,
  check_post_check = TRUE,
  test_method = NULL,
  ...,
```

```
  omnibus = c("no", "all_sig", "at_least_one_sig", "at_least_k_sig", "total"),
  at_least_k = 1,
  p_adjust_method = "none",
  fit_name = "fit",
  get_map_names = FALSE,
  get_test_name = FALSE
)
```

Arguments

fit	The fit object, to be passed to manymome::indirect_effect().
x	The name of the x-variable, the predictor.
m	Must be a list of character vectors. Each character vector stores the name(s) of mediator(s) along a path. The path moves from the first mediator in the vector to the last mediator in the vector. If NULL, the stored paths will be used, which are all the indirect paths in the model between x and y, by default.
y	The name of the y-variable, the outcome variable.
mc_ci	Logical. If TRUE, the default, Monte Carlo confidence intervals will be formed. This argument and boot_ci cannot be both TRUE.
mc_out	The pre-generated Monte Carlo estimates generated by manymome::do_mc, stored in a power4test object. Users should not set this argument and should let power4test() to set it automatically.
boot_ci	Logical. If TRUE, the default, nonparametric bootstrap confidence intervals will be formed. This argument and mc_ci cannot be both TRUE.
boot_out	The pre-generated bootstrap estimates generated by manymome::do_boot, stored in a power4test object. Users should not set this argument and should let power4test() to set it automatically.
check_post_check	Logical. If TRUE, the default, and the model is fitted by lavaan, the test will be conducted only if the model passes the post.check conducted by lavaan::lavInspect() (with what = "post.check").
test_method	The method to do the test. If "ci", then the confidence interval (e.g., bootstrap confidence interval) will be used to do the test. If "pvalue", then asymmetric *p*-value by Asparouhov & Muthén (2021) will be used to do the test, and the confidence interval will not be computed. If NULL, its value will be set to "pvalue" if the number of simulated/bootstrap samples (R) is a value that is supported by the method by Boos and Zhang (2000), and set to "ci" otherwise.
...	Additional arguments to be passed to manymome::many_indirect_effects().
omnibus	If "no", the default, then the test results for all paths are stored. If "all_sig", then only one row of test is stored, and the test is declared significant if *all* paths are significant. If "at_least_one_sig", then only one row of test is stored, and the test is declared significant if at least one of the paths is significant. If "at_least_k_sig", then only one row of test is stored, and the test is declared significant if at least k of the paths is significant, k determined by the argument at_least_k. If "total", then the total indirect effect will be tested.

at_least_k The minimum number of paths required to be significant for the omnibus test to be considered significant. Used when omnibus is "at_least_k_sig".

p_adjust_method
The method to be passed to p.adjust() to adjust the *p*-values when testing the effects. Default is "none" and the *p*-values will not be adjusted. The unadjusted *p*-values will be stored in the column pvalue_org.

fit_name The name of the model fit object to be extracted. Default is "fit". Used only when more than one model is fitted in each replication. This should be the name of the model on which the test is to be conducted.

get_map_names Logical. Used by power4test() to determine how to extract stored information and assign them to this function. Users should not use this argument.

get_test_name Logical. Used by power4test() to get the default name of this test. Users should not use this argument.

Details

This function is to be used in power4test() for testing an indirect effect, by setting it to the test_fun argument.

It uses manymome::many_indirect_effects() to do the test. It can be used on models fitted by lavaan::sem() or fitted by a sequence of calls to stats::lm(), although only nonparametric bootstrap confidence interval is supported for models fitted by regression using stats::lm().

It also supports testing the total indirect effect. Just set omnibus to "total".

Value

In its normal usage, it returns a data frame with the following columns:

- est: The estimated indirect effect for each path.
- cilo and cihi: The lower and upper limits of the confidence interval (95% by default), respectively, for each indirect effect
- sig: Whether a test by confidence interval is significant (1) or not significant (0).
- test_label: A column of labels generated to label the indirect effects.

If omnibus is "all_sig" or "at_least_one"sig", then the data frame has only one row, and the columns "est", "cilo", and "cihi" are NA. The column sig is determined by whether all paths are significant ("all_sig") or whether at least one path is significant ("at_least_one_sig").

See Also

power4test()

Examples

```
# Specify the model

model_simple_med <-
"
m1 ~ x
```

```
m2 ~ x
y ~ m1 + m2 + x
"

# Specify the population values

model_simple_med_es <-
"
y ~ m1: s
m1 ~ x: m
y ~ m2: s
m2 ~ x: l
y ~ x: n
"

# Simulate the data

sim_only <- power4test(nrep = 5,
                       model = model_simple_med,
                       pop_es = model_simple_med_es,
                       n = 100,
                       R = 100,
                       do_the_test = FALSE,
                       iseed = 1234)

# Do the test in each replication

test_ind <- power4test(object = sim_only,
                       test_fun = test_k_indirect_effects,
                       test_args = list(x = "x",
                                        y = "y",
                                        mc_ci = TRUE))
print(test_ind,
      test_long = TRUE)

# Set omnibus = "all_sig" to declare
# significant only if all paths are
# significant

test_ind_all_sig <- power4test(
                       object = sim_only,
                       test_fun = test_k_indirect_effects,
                       test_args = list(x = "x",
                                        y = "y",
                                        mc_ci = TRUE,
                                        omnibus = "all_sig"))
print(test_ind_all_sig,
      test_long = TRUE)
```

`test_moderation` *Test All Moderation Effects*

Description

Test all moderation effects by testing all product terms for a power4test object.

Usage

```
test_moderation(
  fit = fit,
  standardized = FALSE,
  check_post_check = TRUE,
  ...,
  p_adjust_method = "none",
  fit_name = "fit",
  get_map_names = FALSE,
  get_test_name = FALSE
)
```

Arguments

fit	The fit object, to be passed to lavaan::parameterEstimates(), lavaan::standardizedSolution(), or lmhelprs::lm_list_to_partable().
standardized	Logical. If TRUE, lavaan::standardizedSolution() will be used. Can be used only with models fitted by lavaan.
check_post_check	Logical. If TRUE, the default, and the model is fitted by lavaan, the test will be conducted only if the model passes the post.check conducted by lavaan::lavInspect() (with what = "post.check").
...	Additional arguments to be passed to lavaan::parameterEstimates(), lavaan::standardizedSolut or lmhelprs::lm_list_to_partable().
p_adjust_method	The method to be passed to p.adjust() to adjust the *p*-values when testing the effects. Default is "none" and the *p*-values will not be adjusted. The unadjusted *p*-values will be stored in the column pvalue_org.
fit_name	The name of the fit results for which the parameter names will be displayed. Default is "fit".
get_map_names	Logical. Used by power4test() to determine how to extract stored information and assign them to this function. Users should not use this argument.
get_test_name	Logical. Used by power4test() to get the default name of this test. Users should not use this argument.

Details

This function is to be used in power4test() for testing all product terms, by setting it to the test_fun argument.

It is just a wrapper to test_parameters(). It will first identifies all product terms (terms with : in the names), and then call test_parameters(), with pars set to select the regression coefficients of these terms.

Value

In its normal usage, it returns the output returned by `lavaan::parameterEstimates()` or `lmhelprs::lm_list_to_partabl` with the following modifications:

- `est`: The parameter estimates, even if standardized estimates are requested (not `est.std`).
- `cilo` and `cihi`: The lower and upper limits of the confidence interval (95% by default), respectively (not `ci.lower` and `ci.upper`).
- `sig`: Whether a test by confidence interval is significant (`1`) or not significant (`0`).
- `test_label`: A column of labels generated by `lavaan::lav_partable_labels()`, which are usually the labels used by `coef()` to label the parameters.

See Also

`power4test()`, `test_parameters()`

Examples

```
# Specify the model

mod <-
"
m ~ x + w1 + x:w1
y ~ m + x
"

# Specify the population values

mod_es <-
"
m ~ x: n
y ~ x: m
m ~ w1: n
m ~ x:w1: l
"

# Simulate the data

sim_only <- power4test(nrep = 4,
                       model = mod,
                       pop_es = mod_es,
                       n = 100,
                       do_the_test = FALSE,
                       iseed = 1234)

# Do the test in each replication

test_out <- power4test(object = sim_only,
                       test_fun = test_moderation)

print(test_out,
      test_long = TRUE)
```

test_parameters *Test All Free Parameters*

Description

Test all free parameters, including user-defined parameters, for a `power4test` object.

Usage

```
test_parameters(
  fit = fit,
  standardized = FALSE,
  pars = NULL,
  op = NULL,
  remove.nonfree = TRUE,
  check_post_check = TRUE,
  exclude_var = FALSE,
  compare_groups = FALSE,
  ...,
  omnibus = c("no", "all_sig", "at_least_one_sig", "at_least_k_sig"),
  at_least_k = 1,
  p_adjust_method = "none",
  fit_name = "fit",
  get_map_names = FALSE,
  get_test_name = FALSE
)

find_par_names(object, fit_name = "fit")
```

Arguments

`fit`	The fit object, to be passed to `lavaan::parameterEstimates()`, `lavaan::standardizedSolution()`, or `lmhelprs::lm_list_to_partable()`.
`standardized`	Logical. If `TRUE`, `lavaan::standardizedSolution()` will be used. Can be used only with models fitted by `lavaan`.
`pars`	Optional. If set to a character vector, only parameters with `test_label` equal to values in `pars` will be returned. See the help page on valid names.
`op`	Optional. If set to a character vector, only parameters with operators (e.g., `"~"`, `"=~"`) will be returned. If both `pars` and `op` are specified, only parameters meeting *both* requirements will be returned.
`remove.nonfree`	Logical. If `TRUE`, the default, only free parameters will be returned. Ignored if `standardized` is `TRUE` or if the model is not fitted by `lavaan`.

check_post_check
: Logical. If TRUE, the default, and the model is fitted by lavaan, the test will be conducted only if the model passes the post.check conducted by lavaan::lavInspect() (with what = "post.check").

exclude_var
: Logical. If TRUE, exclude variances and error variances from the test.

compare_groups
: Logical. If TRUE, the likelihood ratio test (by lavaan::lavTestLRT()) will be used to test the pairwise-equality constraints for all selected free parameters that appear in all groups. These tests will be reported instead of the tests for individual parameters.

...
: Additional arguments to be passed to lavaan::parameterEstimates(), lavaan::standardizedSolut
or lmhelprs::lm_list_to_partable().

omnibus
: If "no", the default, then the test results for all paths are stored. If "all_sig", then only one row of test is stored, and the test is declared significant if *all* paths are significant. If "at_least_one_sig", then only one row of test is stored, and the test is declared significant if at least one of the paths is significant. If "at_least_k_sig", then only one row of test is stored, and the test is declared significant if at least k of the paths is significant, k determined by the argument at_least_k. If "total", then the total indirect effect will be tested.

at_least_k
: The minimum number of paths required to be significant for the omnibus test to be considered significant. Used when omnibus is "at_least_k_sig".

p_adjust_method
: The method to be passed to p.adjust() to adjust the *p*-values when testing the effects. Default is "none" and the *p*-values will not be adjusted. The unadjusted *p*-values will be stored in the column pvalue_org.

fit_name
: The name of the fit results for which the parameter names will be displayed. Default is "fit".

get_map_names
: Logical. Used by power4test() to determine how to extract stored information and assign them to this function. Users should not use this argument.

get_test_name
: Logical. Used by power4test() to get the default name of this test. Users should not use this argument.

object
: A power4test object.

Details

This function is to be used in power4test() for testing all free and user-defined model parameters, by setting it to the test_fun argument.

For models fitted by lavaan, it uses lavaan::parameterEstimates() to do the test. If bootstrapping was requested (by setting se = "boot"), then it supports bootstrap confidence intervals returned by lavaan::parameterEstimates().

It has preliminary, though limited, supported for models fitted by stats::lm() (through lmhelprs::many_lm()). Tests are conducted by ordinary least squares confidence intervals based on the *t* statistic, reported by stats::confint() applied to the output of stats::lm().

Value

In its normal usage, it returns the output returned by `lavaan::parameterEstimates()` or `lmhelprs::lm_list_to_partabl` with the following modifications:

- `est`: The parameter estimates, even if standardized estimates are requested (not `est.std`).
- `cilo` and `cihi`: The lower and upper limits of the confidence interval (95% by default), respectively (not `ci.lower` and `ci.upper`).
- `sig`: Whether a test by confidence interval is significant (`1`) or not significant (`0`).
- `test_label`: A column of labels generated by `lavaan::lav_partable_labels()`, which are usually the labels used by `coef()` to label the parameters.

Find the names of parameters

To use the argument `pars`, the names as appeared in the function `coef()` must be used. For the output of `lavaan`, this can usually be inferred from the parameter syntax (e.g., `y~x`, no space). If not sure, call `coef()` on the output of `lavaan`. If a parameter is labelled, then the label should be used in `par`.

If not sure, the function `find_par_names()` can be used to find valid names.

See Also

`power4test()`

Examples

```
# Specify the model

mod <-
"
m ~ x
y ~ m + x
"


# Specify the population values

mod_es <-
"
y ~ m: l
m ~ x: m
y ~ x: n
"


# Simulate the data

sim_only <- power4test(nrep = 2,
                       model = mod,
                       pop_es = mod_es,
                       n = 100,
                       do_the_test = FALSE,
                       iseed = 1234)
```

```
# Do the tests in each replication

test_out <- power4test(object = sim_only,
                       test_fun = test_parameters)

print(test_out,
      test_long = TRUE)

# Do the tests in each replication: Standardized solution
# Delta method SEs will be used to do the tests

test_out <- power4test(object = sim_only,
                       test_fun = test_parameters,
                       test_args = list(standardized = TRUE))

print(test_out,
      test_long = TRUE)

# Do the tests in each replication: Parameters with the selected operator

test_out <- power4test(object = sim_only,
                       test_fun = test_parameters,
                       test_args = list(op = "~"))

print(test_out,
      test_long = TRUE)


# Finding valid parameter names

find_par_names(sim_only)
```

---

x_from_power *Sample Size and Effect Size Determination*

Description

It searches by simulation the sample size (given other factors, such as effect sizes) or effect size (given other factors, such as sample size) with power to detect an effect close to a target value.

Usage

```
x_from_power(
  object,
  x = arg_x_from_power(object, "x", arg_in = "call") %||% "n",
  pop_es_name = arg_x_from_power(object, "pop_es_name", arg_in = "call"),
  target_power = 0.8,
  what = arg_x_from_power(object, "what") %||% "point",
```

```
  goal = arg_x_from_power(object, "goal") %||% {
     switch(what, point = "ci_hit", ub
    = "close_enough", lb = "close_enough")
 },
  ci_level = 0.95,
  tolerance = NULL,
  x_interval = switch(x, n = c(50, 2000), es = NULL),
  extendInt = NULL,
  progress = TRUE,
  simulation_progress = NULL,
  max_trials = NULL,
  final_nrep = attr(object, "args")$nrep %||% (object$nrep_final %||% 400),
  final_R = attr(object, "args")$R %||% (object$args$final_R %||% 1000),
  seed = NULL,
  x_include_interval = FALSE,
  check_es_interval = TRUE,
  power_curve_args = list(power_model = NULL, start = NULL, lower_bound = NULL,
    upper_bound = NULL, nls_control = list(), nls_args = list()),
  save_sim_all = FALSE,
  algorithm = NULL,
  control = list(),
  internal_args = list(),
 rejection_rates_args = list(collapse = "all_sig", at_least_k = 1, p_adjust_method =
    "none", alpha = 0.05)
)

n_from_power(
  object,
  pop_es_name = NULL,
  target_power = 0.8,
  what = formals(x_from_power)$what,
  goal = formals(x_from_power)$goal,
  ci_level = 0.95,
  tolerance = NULL,
  x_interval = c(50, 2000),
  extendInt = NULL,
  progress = TRUE,
  simulation_progress = NULL,
  max_trials = NULL,
  final_nrep = formals(x_from_power)$final_nrep,
  final_R = formals(x_from_power)$final_R,
  seed = NULL,
  x_include_interval = FALSE,
  check_es_interval = TRUE,
  power_curve_args = list(power_model = NULL, start = NULL, lower_bound = NULL,
    upper_bound = NULL, nls_control = list(), nls_args = list()),
  save_sim_all = FALSE,
  algorithm = NULL,
```

```
  control = list(),
  internal_args = list(),
 rejection_rates_args = list(collapse = "all_sig", at_least_k = 1, p_adjust_method =
    "none", alpha = 0.05)
)

n_region_from_power(
  object,
  pop_es_name = NULL,
  target_power = 0.8,
  ci_level = 0.95,
  tolerance = NULL,
  x_interval = c(50, 2000),
  extendInt = NULL,
  progress = TRUE,
  simulation_progress = NULL,
  max_trials = NULL,
  final_nrep = formals(x_from_power)$final_nrep,
  final_R = formals(x_from_power)$final_R,
  seed = NULL,
  x_include_interval = FALSE,
  check_es_interval = TRUE,
  power_curve_args = list(power_model = NULL, start = NULL, lower_bound = NULL,
    upper_bound = NULL, nls_control = list(), nls_args = list()),
  save_sim_all = FALSE,
  algorithm = NULL,
  control = list(),
  internal_args = list(),
 rejection_rates_args = list(collapse = "all_sig", at_least_k = 1, p_adjust_method =
    "none", alpha = 0.05)
)

## S3 method for class 'x_from_power'
print(x, digits = 3, call = TRUE, ...)

## S3 method for class 'n_region_from_power'
print(x, digits = 3, ...)

arg_x_from_power(object, arg, arg_in = NULL)

pba_diagnosis(
  a_out,
  p_interval = c(0.05, 0.95),
  posterior_xlim = c(0.01, 0.99),
  which = NULL
)
```

Arguments

object	A power4test object, which is the output of power4test(). Can also be a power4test_by_n object, the output of power4test_by_n(), or a power4test_by_es object, the output of power4test_by_es(). For these two types of objects, the attempt with power closest to the target_power will be used as object, and all other attempts in them will be included in the estimation of subsequent attempts and the final output. Last, it can also be the output of a previous call to x_from_power(), and the stored trials will be retrieved.
x	For x_from_power(), x set the value to be searched. Can be "n", the sample size, or "es", the population value of a parameter (set by pop_es_name). For the print method of x_from_power objects, this is the output of x_from_power().
pop_es_name	The name of the parameter. Required if x is "es". See the help page of ptable_pop() on the names for the argument pop_es.
target_power	The target power, a value greater than 0 and less than one.
what	The value for which is searched: the estimate power ("point"), the upper bound of the confidence interval ("ub"), or the lower bound of the confidence interval ("lb").
goal	The goal of the search. If "ci_hit", then the goal is to find a value of x with the confidence interval of the estimated power including the target power. If "close_enough", then the goal is to find a value of x with the value in what "close enough" to the target power, defined by having an absolute difference with the target power less than tolerance.
ci_level	The level of confidence of the confidence intervals computed for the estimated power. Default is .95, denoting 95%.
tolerance	Used when the goal is "close_enough". If NULL, set automatically based on the algorithm used.
x_interval	A vector of two values, the minimum value and the maximum values of x, in the search for the values (sample sizes or population values). If NULL, default when x = "es", it will be determined internally.
extendInt	Whether x_interval can be expanded when estimating the the values to try. The value will be passed to the argument of the same name in stats::uniroot(). If x is "n", then the default value is "upX". That is, a value higher than the maximum in x_interval is allowed, if predicted by the tentative model. Otherwise, the default value is "no". See the help page of stats::uniroot() for further information.
progress	Logical. Whether the searching progress is reported.
simulation_progress	Logical. Whether the progress in each call to power4test(), power4test_by_n(), or power4test_by_es() is shown. To be passed to the progress argument of these functions. If NULL, set automatically based on the algorithm used.
max_trials	The maximum number of trials in searching the value with the target power. Rounded up if not an integer. If NULL, set automatically based on the algorithm used.

`final_nrep` The number of replications in the final stage, also the maximum number of replications in each call to `power4test()`, `power4test_by_n()`, or `power4test_by_es()`. If `object` is an output of `power4test()` or `x_from_power()` and this argument is not set, `final_nrep` will be set to `nrep` or `final_nrep` stored in `object`.

`final_R` The number of Monte Carlo simulation or bootstrapping samples in the final stage. The `R` in calling `power4test()`, `power4test_by_n()`, or `power4test_by_es()` will be stepped up to this value when approaching the target power. Do not need to be very large because the goal is to estimate power by replications, not for high precision in one single replication. If `object` is an output of `power4test()` or `x_from_power()` and this argument is not set, `final_R` will be set to `R` or `final_R` stored in `object`.

`seed` If not `NULL`, `set.seed()` will be used to make the process reproducible. This is not always possible if many stages of parallel processing is involved.

`x_include_interval`
Logical. Whether the minimum and maximum values in `x_interval` are mandatory to be included in the values to be searched.

`check_es_interval`
If `TRUE`, the default, and `x` is `"es"`, a conservative probable range of valid values for the selected parameter will be determined, and it will be used instead of `x_interval`. If the range spans both positive and negative values, only the interval of the same sign as the population value in `object` will be used.

`power_curve_args`
A named list of arguments to be passed `power_curve()` when estimating the relation between power and `x` (sample size or effect size). Please refer to `power_curve()` on available arguments. There is one except: `power_model` is mapped to the `formula` argument of `power_curve()`.

`save_sim_all` If `FALSE`, the default, the data in each `power4test` object for each value of `x` is not saved, to reduce the size of the output. If set to `TRUE`, the size of the output can be very large in size.

`algorithm` The algorithm for finding `x`. Can be `"power_curve"`, `"bisection"`, or `"probabilistic_bisection"`. The default algorithm depends on `x`.

`control` A named list of additional arguments to be passed to the algorithm to be used. For advanced users.

`internal_args` A named list of internal arguments. For internal testing. Do not use it.

`rejection_rates_args`
Argument values to be used when `rejection_rates()` is called, used to decide how rejection rates will be estimated. Only one single test is supported by `x_from_power()`. Therefore, `merge_all_tests` is always `TRUE` and cannot be changed. The argument `collapse` can be `"all_sig"`, `"at_least_one_sig"`, or `"at_least_k_sig"` (it cannot be `"none"`). Please refer to `rejection_rates()` for other possible arguments. These values, if set, will overwrite any stored settings in `object`.

`digits` The number of digits after the decimal when printing the results.

`call` Logical. Whether the call is printed.

`...` Optional arguments. Not used for now.

arg	The name of element to retrieve.
arg_in	The name of the element from which an element is to be retrieved.
a_out	The output of x_from_power() and friends. The algorithm used must be "probabilistic_bisection".
p_interval	The range of the plot that "zooms" around the solution (or the median of in the final posterior probability distribution), expressed in terms of the area of the distribution.
posterior_xlim	The range of the first plot of search history and the plot of posterior probability distribution, expressed in terms of the area of the distribution.
which	If a_out is a list with more than one output of x_from_power(), such as the output of n_region_from_power(), which must be set to the name of one such output ("below" or "above" for the output of n_region_from_power(), or the output of q_power_mediation_*).

Details

This is how to use x_from_power():

- Specify the model by power4test(), with do_the_test = FALSE, and set the magnitude of the effect sizes to the minimum levels to detect.
- Add the test using power4test() using test_fun and test_args (see the help page of power4test() for details). Run it on the starting sample size or effect size.
- Call x_from_power() on the output of power4test() returned from the previous step. This function will iteratively repeat the analysis on either other sample sizes, or other values for a selected model parameter (the effect sizes), trying to achieve a goal (goal) for a value of interest (what).

If the goal is "ci_hit", the search will try to find a value (a sample size, or a population value of the selected model parameter) with a power level close enough to the target power, defined by having its confidence interval for the power including the target power.

If the goal is "close_enough", then the search will try to find a value of x with its level of power ("point"), the upper bound of the confidence interval for this level of power ("ub"), or the lower bound of the confidence interval fro this level of power ("lb") "close enough" to the target level of power, defined by having an absolute difference less than the tolerance.

If several values of x (sample size or the population value of a model parameter) have already been examined by power4test_by_n() or power4test_by_es(), the output of these two functions can also be used as object by x_from_power().

Usually, the default values of the arguments should be sufficient.

The results can be viewed using summary(), and the output has a plot method (plot.x_from_power()) to plot the relation between power and values (of x) examined.

A detailed illustration on how to use this function for sample size can be found from this page:

https://sfcheung.github.io/power4mome/articles/x_from_power_for_n.html

The function n_from_power() is just a wrapper of x_from_power(), with x set to "n".

The function n_region_from_power() is just a wrapper of x_from_power(), with x set to "n", with two passes, one with what = "ub" and one with what = "lb".

The print method only prints basic information. Call the summary method of x_from_power objects (summary.x_from_power()) and its print method for detailed results

The function arg_x_from_power() is a helper to set argument values if object is an output of x_from_power() or similar functions.

The function pba_diagnosis() generates simple diagnostic plots for the search history of probabilistic bisection algorithm. This function is for advanced users to examine the search history of the probabilistic bisection algorithm. It is for diagnostic purpose and has limited support for customizing the plots.

Value

The function x_from_power() returns an x_from_power object, which is a list with the following elements:

- power4test_trials: The output of power4test_by_n() for all sample sizes examined, or of power4test_by_es() for all population values of the selected parameter examined.
- rejection_rates: The output of rejection_rates().
- x_tried: The sample sizes or population values examined.
- power_tried: The estimated rejection rates for all the values examined.
- x_final: The sample size or population value in the solution. NA if a solution not found.
- power_final: The estimated power of the value in the solution. NA if a solution not found.
- i_final: The position of the solution in power4test_trials. NA if a solution not found.
- ci_final: The confidence interval of the estimated power in the solution. The method is determined by the option power4mome.ci_method. If NULL or "wilson", Wilson's (1927) method is used. If "norm", normal approximation is used.
- ci_level: The level of confidence of ci_final.
- nrep_final: The number of replications (nrep) when estimating the power in the solution.
- power_curve: The output of power_curve() when estimating the power curve.
- target_power: The requested target power.
- power_tolerance: The allowed difference between the solution's estimated power and the target power. Determined by the number of replications and the level of confidence of the confidence intervals.
- x_estimated: The value (sample size or population value) with the target power, estimated by power_curve. This is used, when solution not found, to determine the range of the values to search when calling the function again.
- start: The time and date when the process started.
- end: The time and date when the process ended.
- time_spent: The time spent in doing the search.
- args: A named list of the arguments of x_from_power() used in the search.
- call: The call when this function is called.

The function n_region_from_power() returns a named list of two output of n_from_power(), of the class n_region_from_power. The output with what = "ub" is named "below", and the output with what = "lb" is namd "above".

The print-method of x_from_power objects returns the object x invisibly. It is called for its side effect.

The print-method of x_from_power_region objects returns the object x invisibly. It is called for its side effect.

The function arg_x_from_power() returns the requested argument if available. If not available, it returns NULL.

The function pba_diagnosis() returns NULL invisibly. Called for its side-effect.

Algorithms

Three algorithms are currently available, the simple (though sometimes inefficient) bisection method, a method that makes use of the estimated crude power curve, and probabilistic bisection algorithm (Waeber et al., 2013; see Chalmers, 2024, for applying this algorithm to power analysis).

Unlike typical root-finding problems, the prediction of the level of power is stochastic. Moreover, the computational cost is high when Monte Carlo or bootstrap confidence intervals are used to do a test because the estimation of the power for one single value of x can sometimes take one minute or longer. Therefore, in addition to the simple bisection method, two methods, named *power curve* method (which belongs to the family of surrogate function approximation method reviewed in Chalmers, 2024) and probabilistic bisection method (Chalmers, 2024; Waeber et al., 2013) were also developed for this scenario.

Bisection Method:

This method (called informat bisection in Chalmers, 2024), enabled by algorithm = "bisection", basically starts with an interval that probably encloses the value of x that meets the goal, and then successively narrows this interval. A point inside this interval, usually the mid-point but can also be approximated by another method (e.g., the power curve method), is used as the estimate. Though simple, there are cases in which it can be slow. Nevertheless, preliminary examination suggests that this method is good enough for scenarios in which only an approximate sample size or range of sample sizes is needed.

The internal workflow of this method implemented in x_from_power() can be found in this technical vignette: https://sfcheung.github.io/power4mome/articles/x_from_power_workflow_bisection.html.

Power Curve Method:

This method (belongs to the surrogate function approximation family reviewed in Chalmers, 2024), enabled by algorithm = "power_curve", starts with a crude power curve based on a few points. This tentative model is then used to suggest the values to examine in the next iteration. Unlike some other implementations of this family of methods, the form, not just the parameters, of the model can change across iterations, as more and more data points are available.

This method is the default method for some scenarios, such as x = "es" with goal = "ci_hit" because the relation between the power and the population value of a parameter varies across parameters, unlike the relation between power and sample size, which is monotonic. Therefore, taking into account the working power curve may help finding the desired value of x.

Before version 0.1.1.33, this method can be used only with the goal "ci_hit". Since version 0.1.1.34, it supports all goals, like the bisection method.

The internal workflow of this method implemented in x_from_power() can be found in this technical vignette: https://sfcheung.github.io/power4mome/articles/x_from_power_workflow_power_curve.html.

Probabilistic Bisection:

This method, proposed by Waeber and others (2013) for stochastic root-solving, has been adapted by Chalmers (2024) for power analysis. Similar to bisection, this method starts with an interval, with a initial probability for each value (or range of values), such as sample sizes, as the sample size with the target power. A value is then selected to estimate power (the median, by default). Based on the estimated power for this value, the distribution of probabilities is updated. This process is repeated until some termination criteria are met.

Unlike bisection, each iteration can be conducted with a small number of replications (e.g., 50). The method accumulate evidence in a Bayesian approach, and so the certainty of the solution is based on the accumulation of evidence from successive iterations, not on having strong evidence from a few iterations.

In x_from_power, the version of probabilistic bisection proposed by Waeber et al. (2013) was implemented, with minor changes.

Most importantly, when some termination criteria are met, the candidate value of x (e.g., a sample size) will be checked using a larger number of replications (set by final_nrep) to ensure that the estimated power is indeed close enough to the target power (based on tolerance value, determined internally but can also be set directly). If yes, it will be returned as the solution. If no, then the search will continue, until a maximum number of candidate values has been checked.

Although the bisection method can be fast in some situations (e.g., when the interval is narrow and the solution happens to be inside the interval), the probabilistic bisection method is nearly guaranteed to converge to the solution (as long as the solution is inside the interval). Therefore, this is the default algorithm in some scenarios.

The internal workflow of this method implemented in x_from_power() can be found in this technical vignette: https://sfcheung.github.io/power4mome/articles/x_from_power_workflow_pba.html.

References

Chalmers, R. P. (2024). Solving variables with Monte Carlo simulation experiments: A stochastic root-solving approach. *Psychological Methods*. Advance online publication. doi:10.1037/met0000689

Waeber, R., Frazier, P. I., & Henderson, S. G. (2013). Bisection search with noisy responses. *SIAM Journal on Control and Optimization*, *51*(3), 2261–2279. doi:10.1137/120861898

Wilson, E. B. (1927). Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association, 22*(158), 209-212. doi:10.1080/01621459.1927.10502953

See Also

power4test(), power4test_by_n(), and power4test_by_es().

Examples

```
# Specify the population model

mod <-
"
m ~ x
y ~ m + x
"

# Specify the population values

mod_es <-
"
m ~ x: m
y ~ m: l
y ~ x: n
"

# Generate the datasets

sim_only <- power4test(nrep = 5,
                       model = mod,
                       pop_es = mod_es,
                       n = 100,
                       do_the_test = FALSE,
                       iseed = 2345)

# Do a test

test_out <- power4test(object = sim_only,
                       test_fun = test_parameters,
                       test_args = list(pars = "m~x"))

# Determine the sample size with a power of .80 (default)

# In real analysis, to have more stable results:
# - Use a larger final_nrep (e.g., 400).

# If the default values are OK, this call is sufficient:
# power_vs_n <- x_from_power(test_out,
#                            x = "n",
#                            seed = 4567)
power_vs_n <- x_from_power(test_out,
                           x = "n",
                           progress = TRUE,
                           target_power = .80,
                           final_nrep = 5,
                           max_trials = 1,
                           seed = 1234)
summary(power_vs_n)
plot(power_vs_n)
```

Index