Package 'pilotr'

November 22, 2026

Title Simulate Experimental and Behavioural Data from a Portable Design Specification

Version 0.4.0

Description Generative simulation of experimental and behavioural data sets from a portable JavaScript Object Notation (JSON) design specification shared with the 'Python' package of the same name.
Supports user-specified fixed effect sizes, crossed by-subject and by-item random intercepts and slopes, predictors measured with error, realistic response families (Gaussian, lognormal, shifted lognormal, ex-Gaussian, Bernoulli, Poisson, ordinal and Beta) and simulation-based power and precision-based design analysis, including the Type S and Type M errors of Gelman and Carlin (2014) <doi:10.1177/1745691614551642> and a region of practical equivalence. A shared cross-language random-number generator means that, given the same specification and seed, the R and 'Python' implementations produce identical data wherever they share a maths library, as on Linux, except in a rare Beta case under R 4.6.0 or later that the cross-language vignette describes. Elsewhere, a value that passes through exp(), log() or pow(), whose rounding the IEEE-754 standard does not fix, can differ in its last bits.

License MIT + file LICENSE

URL https://pablobernabeu.github.io/pilotr/r/, https://github.com/pablobernabeu/pilotr

BugReports https://github.com/pablobernabeu/pilotr/issues

Encoding UTF-8

Language en-GB

Depends R (>= 4.0.0)

Imports jsonlite, parallel, stats

Suggests shiny, future, promises, ggplot2 (>= 3.4.0), lme4, lmerTest, callr, knitr, rmarkdown, testthat (>= 3.1.5), MASS, spelling, codetools

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Pablo Bernabeu [aut, cre] (ORCID:
<https://orcid.org/0000-0003-1083-2460>)

Maintainer Pablo Bernabeu <pcbernabeu@gmail.com>

Repository CRAN

Date/Publication 2026-11-22 00:40:10 UTC

Contents

pilotr-package . . . 3
as241 . . . 4
brms_bridge . . . 5
build_spec . . . 7
calibrate_response . . . 8
default_response_name . . . 10
design_conditions . . . 11
generate_design_analysis . . . 12
generate_r_script . . . 15
load_spec . . . 16
make_rng . . . 17
model_data . . . 18
model_formula . . . 18
pilotr_example . . . 20
power_curve_mixed . . . 20
power_design . . . 23
power_mixed . . . 25
power_report . . . 30
precision_curve . . . 33
precision_design . . . 35
print.pilotr_bridge . . . 38
print.pilotr_power . . . 39
replicate_seeds . . . 39
response_variance . . . 41
run_app . . . 43
simulate_design . . . 44
solve_curve . . . 45
spec_from_model . . . 48
spec_json . . . 54
sweep_spec . . . 55
target_n . . . 57
validate_spec . . . 58

Index **60**

pilotr-package *pilotr: Simulate Experimental and Behavioural Data from a Portable Design Specification*

Description

Generative simulation of experimental and behavioural data sets from a portable JSON design specification shared with the Python package of the same name. The user specifies the fixed effect sizes and the crossed by-subject and by-item random intercepts and slopes. The response families cover Gaussian, lognormal, shifted lognormal, ex-Gaussian, Bernoulli, Poisson, ordinal and Beta outcomes. Power and precision-based design analysis run from the same specification.

Details

A pilotr workflow begins with a design specification, a plain list recording the study you plan to run: its groups and conditions, sample sizes, fixed effect sizes, random-effect standard deviations and the response family. Assemble one from a flat list of design inputs with `build_spec()`, or read one back from a JSON file with `load_spec()`. The package ships eight ready-to-run specifications, and `pilotr_example()` returns their paths. `default_response_name()` gives the response column that a family uses by default, and `spec_json()` serialises a specification back to JSON for the Python package or the no-code app to read.

`simulate_design()` turns a specification into an analysis-ready data frame with one row per observation. A specification carries its own seed, and both languages draw from the combined generator built by `make_rng()` on top of the inverse-normal routine `as241()`, so a given specification and seed produce identical data in either language wherever the two share a maths library, as on Linux, outside a rare Beta case under R 4.6.0 or later. Elsewhere, a value that passes through `exp()`, `log()` or `pow()` can differ in its last bits, as `vignette("cross-language", package = "pilotr")` sets out.

For analysis, `model_data()` adds the response, contrast and interaction columns that the model expects, and `model_formula()` derives the mixed-model formula the design implies, maximal by default. `brms_bridge()` returns the formula, family and priors for a Bayesian fit.

Design analysis runs from that same specification. `power_design()` estimates power for a two-group Gaussian design with one row per subject, together with the Type S and Type M errors of Gelman and Carlin (2014). `power_mixed()` does the same for a crossed mixed-effects design, and `power_curve_mixed()` sweeps sample size. `target_n()` solves that curve for the sample size at which a design becomes adequately powered, with an interval on it. `precision_design()` and its curve counterpart `precision_curve()` report the width of the interval a design gives and the decision probabilities against a region of practical equivalence.

`generate_r_script()` writes a self-contained script that reproduces a simulation, and `run_app()` launches the bundled no-code app.

For a worked introduction, see `vignette("getting-started", package = "pilotr")`.

Author(s)

Pablo Bernabeu, author and maintainer (<pcbernabeu@gmail.com>, ORCID).

References

Gelman, A. and Carlin, J. (2014). Beyond power calculations: Assessing Type S (sign) and Type M (magnitude) errors. *Perspectives on Psychological Science*, 9(6), 641-651. doi:10.1177/1745691614551642

See Also

Useful links:

- https://pablobernabeu.github.io/pilotr/r/
- https://github.com/pablobernabeu/pilotr
- Report bugs at https://github.com/pablobernabeu/pilotr/issues

---

`as241` *Inverse normal cumulative distribution function (Wichura's AS 241)*

Description

Compute the standard-normal quantile for a probability using Wichura's (1988) Algorithm AS 241 (the PPND16 routine), the same algorithm underlying `stats::qnorm()`.

Usage

```
as241(p)
```

Arguments

`p`	A probability in the open interval (0, 1).

Details

The tail branch, which about 15% of uniform draws reach, calls `log()`. Where R and 'Python' do not share a maths library, the two can therefore return different doubles for the same `p`. On Windows, 14 of 1,000,000 draws differed, by 1 to 3 units in the last place.

Value

The standard-normal quantile (a numeric value) corresponding to `p`.

References

Wichura, M. J. (1988). Algorithm AS 241: The percentage points of the normal distribution. *Applied Statistics*, 37(3), 477-484. doi:10.2307/2347330

Examples

```
as241(0.975)   # about 1.959964
```

brms_bridge *Derive a brms formula, family and priors from a design spec*

Description

Derive a brms formula, family and priors from a design spec

Usage

```
brms_bridge(spec, prior_scale = 0.5, interaction_scale = NULL)
```

Arguments

spec	A design specification (path or list).
prior_scale	The standard deviation of the normal prior on a standardised main effect for the continuous families, that is on the coefficient multiplied by its column's standard deviation and divided by the response's. For bernoulli, poisson, ordinal and beta, it is the standard deviation of the prior on the coefficient itself, on the link scale.
interaction_scale	The same for interaction terms. Defaults to half of prior_scale.

Details

The formula names the analysis columns of the design: each factor's numeric contrast columns and the predictors, with every interaction left for brms to form from its parts. A data set as simulate_design() returns it holds the factors' level labels and not their contrast columns, so it goes through model_data() first, as in model_data(spec, simulate_design(spec)). The emitted call says so beside its data argument. Collected data laid out in the same way go through model_data() too.

brms names an interaction coefficient after its parts in the order they first appear in the formula, as stats::terms() does, so a key cond:age in a specification that lists age first is the coefficient age:cond. The formula and the priors use the name brms uses. A specification that gives one interaction two keys, such as cond:age and age:cond, is refused, since brms estimates a single coefficient for it.

For gaussian and exgaussian responses, the coefficients are on the response's own scale, and for lognormal and shifted_lognormal on the scale of its logarithm, so their priors are scaled to the data the design produces. Take sd_y to be the square root of the total variance that response_variance() reports, and sd_x the standard deviation of a coefficient's column in model_data(spec, simulate_design(spec)), an interaction's product column included. The coefficient's prior is then normal(0, s * sd_y / sd_x), where s is prior_scale for a main effect and interaction_scale for an interaction, and the random-effect standard deviations take a half-normal prior with standard deviation sd_y. A coefficient whose column does not vary in the design cannot be told apart from the intercept. For these families, it draws a warning, and its prior is scaled as if sd_x were 1. For bernoulli, poisson, ordinal and beta, the coefficients are on the logit or log scale, which does not depend on the

response's units. A coefficient's prior is then `normal(0, s)` per unit of its column, and the random-effect standard deviations take `normal(0, 1)`. These widths suit contrast columns and standardised predictors, while a predictor on a wider scale, such as age in years, gets a correspondingly wider prior per standard deviation. Widths are written to four significant digits.

A prior on the random-effect standard deviations is emitted only when the design has random effects, and an LKJ prior on their correlations only when some grouping factor's effects are correlated. brms refuses a prior on a parameter the model does not contain. The intercept and the family's own parameters, such as the residual standard deviation, take brms's default priors. For the continuous families, brms centres its default prior for the intercept on the median of the response, or of its logarithm for the two lognormal families.

A Bayes factor computed against these priors depends on their widths (Kass and Raftery, 1995), so a different `prior_scale` gives a different Bayes factor from the same data. For a continuous family at the default of 0.5, a standardised effect of 0.5 lies one prior standard deviation from zero.

A `beta` response lies strictly inside (0, 1), as `Beta()` requires, because `simulate_design()` moves a draw or a rounded value at exactly 0 or 1 to the nearest value inside the interval. When `mu * phi` or `(1 - mu) * phi` is well below 1, part of the distribution lies nearer to 0 or 1 than doubles resolve. Those rows sit at the bounds, which are `2^-1074` and `1 - 2^-53` for an unrounded response. There `log(y)` or `log(1 - y)` reaches about -744 or -37, so such rows dominate a Beta likelihood. They are an artefact of double precision. The process that generated them has no point mass at the boundary, so a zero-one-inflated model would misstate it.

Value

An object of class `pilotr_bridge`, a list whose four elements are text. `formula` is the model formula, and `family` the brms family call, such as `"shifted_lognormal()"`. `priors` holds the `prior()` calls, one string each, and `code` the lines that load brms and fit the model with `brm()`, ready to copy into a script once `your_data` names the data. The object is returned visibly, and `print.pilotr_bridge()` writes `code` to the console, so a bare call shows the model while an assignment stays silent.

References

Kass, R. E. and Raftery, A. E. (1995). Bayes factors. *Journal of the American Statistical Association*, 90(430), 773-795. doi:10.1080/01621459.1995.10476572

See Also

`print.pilotr_bridge()` for the display, and `model_formula()` for the frequentist counterpart of the formula.

Examples

```
spec <- build_spec(list(name = "d", seed = 1, design_kind = "within",
  include_items = TRUE, n_subject = 20, n_item = 12, factor_name = "cond",
  lev1 = "a", lev2 = "b", intercept = 6, effect = 0.05,
  subj_int_sd = 0.12, subj_slope_sd = 0.04, subj_corr = 0.2,
  item_int_sd = 0.08, item_slope_sd = 0.02, item_corr = -0.1,
  family = "shifted_lognormal", resp_name = "RT", sigma = 0.3, shift = 200))
bridge <- brms_bridge(spec)          # silent
```

```
bridge$formula
bridge                              # prints the ready-to-fit model
```

`build_spec` *Build a design specification from a flat list of design inputs*

Description

Assemble a portable design specification (a plain list, serialisable with `spec_json()`) from the flat set of inputs collected by the no-code application: sample sizes, the two-level factor and its levels, the fixed intercept and effect, the random-effect standard deviations for within-subject and crossed designs and the response family with its parameters.

Usage

```
build_spec(p)
```

Arguments

`p`	A named list of design inputs. Common fields are `name`, `seed`, `n_subject`, `design_kind` (`"between"` or `"within"`), `include_items` and `n_item` (read for a within design only), `factor_name`, `lev1`, `lev2`, `intercept`, `effect`, `family`, `resp_name` (blank or `NULL` for the name `default_response_name()` gives) and the family's parameters: `sigma` for the four continuous families, `shift` for `"shifted_lognormal"`, `beta` for `"exgaussian"`, `phi` for `"beta"` and, for `"ordinal"`, `thresholds` as a single comma-separated string such as `"-2, -0.6, 0.6, 2"`, as the no-code app supplies it. Within-design random effects use `subj_int_sd`, `subj_slope_sd`, `subj_corr`, `item_int_sd`, `item_slope_sd` and `item_corr`. `counterbalance = TRUE` counterbalances the factor of a within design with items, and is ignored in any other design.

Details

A response of the Gaussian, lognormal, shifted lognormal or ex-Gaussian family is rounded to four decimals, and a `beta` response is left unrounded. Rounding moves every proportion near 0 or 1 onto the nearest value inside the interval at that many decimals. At four decimals, 9.8% of the 40,000 rows of a design with `phi = 0.8` landed on 0.0001 or 0.9999, and rows at such a bound dominate a Beta likelihood (see `brms_bridge()`). Unrounded, the data keep the distribution the specification describes. They can then differ between R and 'Python' in their last bits where the two do not share a maths library, as the cross-language vignette sets out.

Counterbalanced lists show each subject each item at one level of the factor, rotating the levels over subjects and items as a Latin square does, so a design has half the rows of the fully crossed one. They set `counterbalance` on the factor, a field of specification version 0.4 described under Counterbalanced lists in the specification.

The specification declares the lowest `spec_version` that reads it. Only counterbalanced lists need a version above 0.3, so the releases of either package that read 0.3 read every other design built here.

Value

A design specification as a nested list, ready for `simulate_design()`, `spec_json()` or the power and precision functions.

Examples

```
build_spec(list(name = "demo", seed = 1, design_kind = "between",
  factor_name = "group", lev1 = "control", lev2 = "treatment", n_subject = 40,
  intercept = 100, effect = 5, family = "gaussian", resp_name = "", sigma = 10))
```

`calibrate_response` *Rescale a design to a target total variance*

Description

Adjust a specification so that the total variance of its linear predictor, plus the residual variance of its response family, comes to `target_var`. Calibrating to 1 puts the outcome on a unit scale, which is what lets a region of practical equivalence or a smallest effect size of interest be stated in standard-deviation units and read the same way across designs. The total includes the variance the fixed effects contribute, so a coefficient calibrated against it is in standard deviations of the outcome and is not Cohen's d. With `denominator = "random"`, the target applies to the total less that fixed component, the variance that d divides by.

Usage

```
calibrate_response(
  spec,
  target_var = 1,
  tune = c("sigma", "all"),
  scale_intercept = FALSE,
  denominator = c("total", "random")
)
```

Arguments

`spec`	A design specification (path or list).
`target_var`	The variance to calibrate to, as `denominator` defines it. Defaults to 1.
`tune`	Either `"sigma"` (the default), which solves for the residual standard deviation alone, or `"all"`, which scales every variance-contributing term by a common factor.
`scale_intercept`	With `tune = "all"`, whether the intercept is scaled with the rest. `FALSE` (the default) holds it, and `TRUE` multiplies it by the same factor as the rest (see Details).
`denominator`	Either `"total"` (the default), the total variance with the fixed component included, or `"random"`, the random-effect components plus the residual, as in Cohen's d.

Details

`tune = "sigma"` holds the fixed effects and the random-effect standard deviations where they are and solves for the residual standard deviation. That keeps every effect size in the specification as written, and is the right choice when those effects come from a pilot study or from the literature. It fails when the structural variance already exceeds the target, since no residual standard deviation can bring the total down, and the error says so rather than returning a negative variance.

`tune = "all"` holds the intercept and multiplies every coefficient and every random-effect standard deviation, and the residual standard deviation where there is one, by a common factor. For the families with a free residual, that leaves every ratio between components unchanged, so it rescales the outcome without altering the design's character. It is also the only option for the families whose residual is fixed by the link.

For those, the residual cannot be rescaled at all, so the structural part alone has to close the gap and the target has to exceed the link's own contribution. A `bernoulli` or `ordinal` design carries a latent residual of `pi^2 / 3`, about 3.29, so it cannot be calibrated to a total variance of 1, and the error says so rather than returning something plausible. For `poisson` and `beta`, the residual moves with the linear predictor, so the factor is solved numerically. That costs one extra simulation rather than one per candidate, because scaling the coefficients and the random-effect standard deviations by `k` moves each row's linear predictor from `eta` to exactly `b0 + k * (eta - b0)`, where `b0` is the intercept. The search starts at `k = 0`, where every row sits at the intercept, so a target at or below the residual variance that the intercept alone implies is refused.

The intercept is held because it is a location. It adds nothing to the fixed or random components, so scaling it changes none of them and only moves the outcome. For `poisson` and `beta`, whose residual depends on the location, scaling the intercept would reach the target partly by moving the mean count or proportion. In the first example below, the median reaction time of 600 ms stays near 600 ms. The intercept is the location when the contrasts and predictors are centred, as codes of -0.5 and 0.5 are. Scaling the coefficient of a contrast coded 0 and 1, or of a predictor whose mean is not zero, moves the location as well.

`scale_intercept = TRUE` multiplies the intercept by the same factor. For `gaussian` and `exgaussian`, that is a change of units, and no power or precision result for a coefficient changes. Through a log or logit link, it changes the outcome itself. For the lognormal families, it raises the response, less any shift, to the power of the factor, which turns that median of 600 ms into 16.6 hours. For `poisson`, it refuses targets that a held intercept reaches, and it can return an intercept too large to simulate.

`denominator` says what `target_var` is a target for. `"total"`, the default, is the total that `response_variance()` reports, fixed component included. `"random"` is that total less the fixed component, leaving the random-effect components and the residual. In a balanced design, that is the variance within a condition. Westfall, Kenny and Judd (2014) define Cohen's d for designs in which participants respond to stimuli by dividing a difference between condition means by the square root of that variance (see also Brysbaert and Stevens, 2018). Once it is calibrated to 1, the coefficient of a two-level factor coded -0.5 and 0.5 is d, on the scale of the linear predictor and conditional on any covariates. The fixed component comes on top, so the total then exceeds `target_var`. With `tune = "sigma"`, the residual standard deviation makes up what the random components leave of the target. `response_variance()` gives the conversion of a published d into a coefficient.

Value

The specification, with the tuned parameters replaced.

References

Brysbaert, M. and Stevens, M. (2018). Power analysis and effect size in mixed effects models: A tutorial. *Journal of Cognition*, 1(1), 9. doi:10.5334/joc.10

Westfall, J., Kenny, D. A. and Judd, C. M. (2014). Statistical power and optimal design in experiments in which samples of participants respond to samples of stimuli. *Journal of Experimental Psychology: General*, 143(5), 2020-2045. doi:10.1037/xge0000014

See Also

`response_variance()` for the decomposition this works from.

Examples

```
spec <- pilotr_example("crossed_mixed_rt")
calibrated <- calibrate_response(spec, target_var = 1, tune = "all")
round(response_variance(calibrated)$total, 6)
calibrated$fixed$intercept   # held at 6, so the median reaction time stays near 600 ms

# Calibrate the random components plus the residual, so that the coefficient of `cond`, coded
# -0.5 and 0.5, is Cohen's d on the log scale.
d_scale <- calibrate_response(spec, target_var = 1, tune = "all", denominator = "random")
d_scale$fixed$coefficients$cond
```

---

default_response_name *Default response-column name for a family*

Description

Default response-column name for a family

Usage

```
default_response_name(family)
```

Arguments

`family`	A response-family name, one of `"gaussian"`, `"lognormal"`, `"shifted_lognormal"`, `"exgaussian"`, `"bernoulli"`, `"poisson"`, `"ordinal"` or `"beta"`.

Value

The conventional response-column name for that family (for example `"RT"` for the reaction-time families `"lognormal"`, `"shifted_lognormal"` and `"exgaussian"`), or `"outcome"` for an unrecognised family.

Examples

```
default_response_name("bernoulli")
```

`design_conditions` *Build a grid of fixed-effect coefficient sets*

Description

Produce the conditions needed to sweep an effect size with `sweep_spec()`, including a condition in which every named effect is zero, so that the same run shows both what a design detects and how often it declares something when there is nothing to find.

Usage

```
design_conditions(..., .null = TRUE, .base = NULL)
```

Arguments

`...`	Named numeric vectors, one per coefficient to vary.
`.null`	Whether to prepend a condition with every named effect set to zero. `TRUE` by default.
`.base`	Optional named list of coefficients to include in every condition, the null condition among them. `sweep_spec()` already keeps the coefficients a condition does not name. `.base` is therefore needed only to hold a coefficient at a value other than the specification's, or for a condition used outside `sweep_spec()`, such as one assigned to `fixed$coefficients` by hand.

Details

Named arguments give the values each effect should take, and are recycled to a common length, so `design_conditions(cond = c(0.02, 0.05), age = 0.1)` produces two conditions, both with age at 0.1. The all-zero condition comes first and is shared, since the Type I error rate is a property of the design rather than of any one effect size.

`sweep_spec()` merges each condition into the specification's own coefficients and keeps their order, so any coefficient not named here keeps its value and a sweep varies only the effects it names. It refuses a condition that names a coefficient the specification lacks. Subsetting the result with `[` keeps its class, and so does `c()` when the result comes first. The three conditions of `c(design_conditions(cond = 0.02), design_conditions(age = 0.1, .null = FALSE))` are therefore merged in the same way. A list assembled by other means replaces the coefficients wholesale.

Value

A list of class `pilotr_conditions` holding one named list of coefficients per condition. The class is how `sweep_spec()` recognises conditions to merge.

See Also

`sweep_spec()`, which consumes this.

Examples

```
design_conditions(effect = c(0.03, 0.06))
design_conditions(cond = c(0.02, 0.05), age = 0.1, .null = FALSE)
```

`generate_design_analysis` *Generate a Bayesian design-analysis script from a specification*

Description

Emit a runnable R script that simulates from a design specification, fits the confirmatory Bayesian model that `brms_bridge()` derives for it and decides about each focal effect on a Savage-Dickey Bayes factor together with a highest-density interval tested against a region of practical equivalence. The script is returned as a character string and nothing is fitted here, so the function needs neither `brms` nor Stan installed.

Usage

```
generate_design_analysis(
  spec,
  focal,
  rule = list(bf = 10, rope = 0.05, ci = 0.95),
  engine = "brms",
  gate = list(max_rhat = 1.01, max_divergent = 0.01),
  array = c("none", "slurm"),
  file = NULL
)
```

Arguments

`spec`	A design specification (path or list).
`focal`	A character vector of focal coefficient names, or a named numeric vector mapping those names to their true values. The names are the specification's coefficients, so an interaction is written `a:b`, not `a_b` as in the `lme4` formula from `model_formula()`, with its parts in either order. The script refers to each effect by the name brms gives it, which for an interaction lists the parts in the order they first appear in the formula, so `cond:age` reaches the script as `age:cond` when the specification lists `age` first. A name that is not a fixed coefficient of the design is reported as a warning here, well before the emitted script would fail on it.
`rule`	The decision rule, a named list with elements `bf` (the Bayes-factor threshold, applied in both directions), `rope` (the half-width of the region of practical equivalence, on the scale of the model's coefficients) and `ci` (the mass of the highest-density interval). A missing element takes its default. Setting `rope` to 0 leaves a verdict of `"null"` unreachable, since no interval of positive width lies inside a region of no width, which turns the rule into a Bayes factor plus a sign requirement.

`engine` The fitting engine for the emitted script. Only `"brms"` is supported.

`gate` The convergence gate, a named list with elements `max_rhat` (the largest acceptable R-hat over all parameters) and `max_divergent` (the largest acceptable share of post-warmup draws ending in a divergent transition). A missing element takes its default.

`array` Either `"none"` for the analysis script alone, or `"slurm"` to append a SLURM array wrapper that runs one replicate per task and an aggregator that combines the per-task results. The three parts are separated by `# ===== FILE n of 3` banners and are meant to be split into three files, since the middle part is shell and so is not valid R. The wrapper carries two placeholders, marked `EDIT` in its header, that must be filled in before submission: the `#SBATCH --account` directive and the writable `PROJECT_DIR`. The partitions it names, `short` in an `#SBATCH --partition` directive and `devel` in the smoke-test command, are cluster-specific, and its header lists them with the placeholders as settings to check and edit. It runs the `design_analysis.R` saved next to it, so submit from the directory the parts were saved into.

`file` Optional path. When given, the script is also written there, byte for byte with LF line endings and a single trailing newline on every platform, and returned invisibly.

Details

The function emits a script for the user to run, for two reasons that both come down to where a Stan model can be built. The no-code application ships as a webR build that runs entirely in the browser, and Stan compiles C++ at fit time, so a Bayesian fit cannot run there at all. A function that produced its analysis only when `brms` was present would then be missing from the build most users meet first. The second reason is maintenance. Depending on `brms` would pull a Stan toolchain into this package's own test and check matrix, and one maintainer cannot keep that working across the platforms CRAN builds on. Emitting a script keeps the analysis reproducible and open to inspection while leaving the fit where a compiler is available.

The verdict rests on two criteria that answer different questions. A Bayes factor compares the null with the alternative and so reports which of the two the data favour (Kass and Raftery, 1995), computed here as the Savage-Dickey density ratio, the ratio of prior to posterior density at zero (Wagenmakers et al., 2010). The interval against a region of practical equivalence asks instead whether the effect is large enough to matter (Kruschke, 2018). Requiring the two to agree makes `"supported"` and `"null"` harder to reach than either criterion alone would, and it leaves the third answer of `"inconclusive"` available when they disagree, which is the answer most small pilots warrant.

Because the Savage-Dickey ratio is read off the prior as well as the posterior, the emitted `brm()` call sets `sample_prior = "yes"`, and the Bayes factor it produces is a statement about the prior that `brms_bridge()` supplies as much as about the data. Widening that prior moves the factor towards the null. The prior is the bridge's at its defaults. For the continuous families, a coefficient's prior has a standard deviation of 0.5 standard deviations of the response per standard deviation of the coefficient's column, and 0.25 for an interaction. For the link families, it is 0.5 and 0.25 on the link scale, per unit of the coefficient's column. Bayes factors depend on these widths (Kass and Raftery, 1995), so they belong in any report of the design analysis.

The convergence gate is checked before any verdict is formed. R-hat is the rank-normalised version of Vehtari et al. (2021), for which a limit near 1.01 is appropriate where the older one was 1.1, and divergent transitions are counted as a share of post-warmup draws so that the threshold means the same thing whatever the run length. When either limit is exceeded, the emitted script reports `NA` for every verdict and prints why, since a fit that has not converged cannot support a conclusion.

A verdict is a function of the whole rule, the whole gate and the specification, so the record of each replicate carries all of them: `bf_threshold`, `rope`, `ci_mass`, `max_rhat_limit`, `max_divergent_limit`, an MD5 fingerprint of the canonical specification JSON and the pilotr version that produced the data. A results directory collects whatever was written into it, and the emitted aggregator globs it, so without those columns two array runs under different regions of practical equivalence would combine into one table with nothing to tell them apart. The aggregator stops when they disagree, and pools nothing.

Value

A length-one character string holding the emitted script, invisibly when `file` is given. For `array = "slurm"`, the string holds three banner-separated parts: an R analysis script, a bash array wrapper and an R aggregator. The wrapper cannot be submitted as emitted, since its `--account` and `PROJECT_DIR` placeholders must be filled in first and its partitions checked against the cluster's.

References

Kass, R. E. and Raftery, A. E. (1995). Bayes factors. *Journal of the American Statistical Association*, 90(430), 773-795. doi:10.1080/01621459.1995.10476572

Kruschke, J. K. (2018). Rejecting or accepting parameter values in Bayesian estimation. *Advances in Methods and Practices in Psychological Science*, 1(2), 270-280. doi:10.1177/2515245918771304

Vehtari, A., Gelman, A., Simpson, D., Carpenter, B. and Bürkner, P.-C. (2021). Rank-normalization, folding, and localization: An improved R-hat for assessing convergence of MCMC. *Bayesian Analysis*, 16(2), 667-718. doi:10.1214/20BA1221

Wagenmakers, E.-J., Lodewyckx, T., Kuriyal, H. and Grasman, R. (2010). Bayesian hypothesis testing for psychologists: A tutorial on the Savage-Dickey method. *Cognitive Psychology*, 60(3), 158-189. doi:10.1016/j.cogpsych.2009.12.001

See Also

`brms_bridge()` for the model the script fits, `precision_design()` for the frequentist analogue of the interval criterion, which runs in place, and `generate_r_script()` for the simulation-only script.

Examples

```
# Emitting the script needs neither brms nor Stan, which is what lets it work in the
# browser build of the no-code app.
spec   <- load_spec(pilotr_example("crossed_mixed_rt"))
script <- generate_design_analysis(spec, focal = c(cond = 0.05))
cat(head(strsplit(script, "\n")[[1]], 18), sep = "\n")

# A stricter rule, with a wrapper for a SLURM array and its aggregator.
cluster <- generate_design_analysis(spec, focal = c(cond = 0.05),
```

```
                                    rule = list(bf = 30, rope = 0.02),
                                    array = "slurm")
cat(grep("^# ===== FILE", strsplit(cluster, "\n")[[1]], value = TRUE), sep = "\n")
```

`generate_r_script` *Generate a self-contained, reproducible R script from a specification*

Description

Embed the specification as the JSON text that `spec_json()` writes, so that the returned script reproduces the design without any external file. This turns a design built in the no-code application into a reproducible script. The application's Verify button runs that script in a clean R session and checks that its data are identical, bit for bit, to the data the application's current specification produces.

Usage

```
generate_r_script(spec)
```

Arguments

`spec`	A design specification (list), as produced by `build_spec()`, or the path of a specification file for `load_spec()` to read.

Details

The script ends with a commented suggestion of the analysis that fits the design. A two-group Gaussian design with one row per subject is sent to `power_design()`, whose t-test takes no other. An `ordinal` or `beta` outcome is sent to `generate_design_analysis()`, which writes a Bayesian design analysis on the link scale to `design_analysis.R`, since `power_mixed()` fits such an outcome only on the response scale. Running that file needs `brms` and a working Stan backend, which the installation line does not name. Any other design is sent to `power_mixed()`, and only then does the installation line name `lme4` and `lmerTest`, which it needs.

Since the script is meant to reproduce the data bit for bit, the embedded specification has to preserve every coefficient exactly. The JSON text sits in a raw string that `load_spec()` reads with 'jsonlite', which parses numbers with the C library's `strtod()`. That rounds correctly, so each number reaches the script as the double it was on every build of R, and the numbers stay as readable as in the specification file. Written as R literals, they would be left to R's own parser. On a build without long-double arithmetic, such as R on Apple silicon, that parser can read a 17-digit decimal one unit in the last place away. `deparse()` is no alternative either, since it prints 15 significant digits and `deparse(1/3)` reads back as a different double.

Value

A length-one character string containing a runnable R script that loads `pilotr`, embeds the specification and simulates the data, with the installation line and the suggested analysis as comments.

Examples

```
spec <- build_spec(list(name = "demo", seed = 1, design_kind = "between",
  factor_name = "group", lev1 = "a", lev2 = "b", n_subject = 20,
  intercept = 0, effect = 0.5, family = "gaussian", resp_name = "", sigma = 1))
cat(generate_r_script(spec))
```

---

load_spec *Load a design specification from a JSON file*

Description

Load a design specification from a JSON file

Usage

```
load_spec(path, validate = TRUE)
```

Arguments

path	Path to a JSON design-specification file, or the JSON text itself, as the scripts from generate_r_script() and generate_design_analysis() embed it.
validate	Whether to validate the specification after reading it. TRUE (the default) applies validate_spec() with strict = TRUE, and FALSE skips validation. Any other value validates leniently, with strict = FALSE, so that an unknown field draws a warning, not an error.

Details

The specification is validated by default, via validate_spec(), because several ways of getting one wrong produce plausible data and no error at all: a mistyped coefficient key resolves to no column and so silently sets that effect to zero, and a response parameter left over from another family is ignored. Validation also refuses a specification declaring a spec_version newer than this implementation understands, and never reads such a file in part.

The file is read as UTF-8, with or without a byte-order mark. A file that repeats a key within one JSON object is refused whatever validate says, because JSON leaves a repeated key undefined and the R and 'Python' readers settle it differently: one coefficient given twice was applied twice at its first value in R and once at its second in 'Python'.

Value

The specification as a nested list, with sub-lists left unsimplified so that the structure round-trips exactly. Pass the result to simulate_design().

Examples

```
spec <- build_spec(list(name = "demo", seed = 1, design_kind = "between",
  factor_name = "group", lev1 = "a", lev2 = "b", n_subject = 20,
  intercept = 0, effect = 0.5, family = "gaussian", resp_name = "", sigma = 1))
f <- tempfile(fileext = ".json")
writeLines(spec_json(spec), f)
identical(simulate_design(load_spec(f)), simulate_design(spec))
```

`make_rng` *Create a shared cross-language random-number generator*

Description

Build the combined linear congruential generator (L'Ecuyer 1988) used by both the R and Python implementations, so that a given specification and seed produce identical data in either language wherever the two share a maths library, except in a rare Beta case under R 4.6.0 or later. The uniforms are exact everywhere. The order of the draws, and where a value can differ in its last bits, are documented in the specification at https://github.com/pablobernabeu/pilotr/blob/main/spec/SPEC.md.

Usage

```
make_rng(seed)
```

Arguments

`seed`	A single number used to seed the generator (coerced to a non-negative integer).

Value

A list of three functions: `uniform()` returns one standard-uniform draw, `normal()` returns one standard-normal draw and `normals(k)` returns a length-k numeric vector of standard-normal draws.

References

L'Ecuyer, P. (1988). Efficient and portable combined random number generators. *Communications of the ACM*, 31(6), 742-751. doi:10.1145/62959.62969

Examples

```
rng <- make_rng(1)
rng$uniform()
rng$normals(3)
```

model_data — *Build the modelling data frame from a simulated data set and its specification*

Description

Add the analysis response column `.y` (`log(y)` for `lognormal`, `log(y - shift)` for `shifted_lognormal` and the response itself otherwise), the numeric contrast columns implied by the categorical factors and any interaction product columns (an `a:b` coefficient becomes a column `a_b`).

Usage

```
model_data(spec, d)
```

Arguments

`spec`	A design specification (path or list). A path is read with `load_spec()`, which validates it.
`d`	A simulated data set, as returned by `simulate_design()`.

Value

The data frame `d` augmented with the `.y` response column and the contrast and interaction columns required by the auto-derived model.

Examples

```
spec <- build_spec(list(name = "d", seed = 1, design_kind = "between",
  factor_name = "group", lev1 = "a", lev2 = "b", n_subject = 20,
  intercept = 0, effect = 0.5, family = "gaussian", resp_name = "", sigma = 1))
head(model_data(spec, simulate_design(spec)))
```

model_formula — *Derive the lmer formula implied by a specification*

Description

Construct the mixed-model formula (with response `.y`, as produced by `model_data()`) from the fixed-effect coefficients and random-effects structure of a specification. Interaction coefficients written `a:b` become formula terms `a_b`.

Usage

```
model_formula(spec, structure = c("maximal", "zcp", "intercepts"))
```

Arguments

spec	A design specification (path or list). A path is read with load_spec(), which validates it.
structure	The random-effects structure: "maximal", the default, "zcp" or "intercepts" (see Details).

Details

The three structures are the usual steps of a rule that simplifies a model when its fit is singular or fails to converge (Bates et al., 2015; Matuschek et al., 2017). Barr et al. (2013) recommend the maximal structure and suggest dropping its correlations when it fails to converge. "maximal" has every random slope the specification declares, correlated with the intercept where the specification correlates them, and uncorrelated (||) where it does not. "zcp", the zero-correlation-parameter model, keeps those slopes and estimates no correlation among them, writing || for every grouping factor with a slope. Since model_data() supplies numeric contrast columns, || does remove the correlations, which it does not for a factor. A grouping factor without slopes keeps (1 | g), since lme4 refuses (1 || g). "intercepts" keeps a random intercept for each grouping factor and drops every slope. power_mixed() and precision_design() take these names as the rungs of such a rule.

A specification with neither a fixed coefficient nor a random effect implies no model, and is refused with the same message as in the 'Python' package.

Value

A stats::formula object suitable for fitting with lme4 or lmerTest.

References

Barr, D. J., Levy, R., Scheepers, C. and Tily, H. J. (2013). Random effects structure for confirmatory hypothesis testing: Keep it maximal. *Journal of Memory and Language*, 68(3), 255-278. doi:10.1016/j.jml.2012.11.001

Bates, D., Kliegl, R., Vasishth, S. and Baayen, H. (2015). Parsimonious mixed models. *arXiv*. doi:10.48550/arXiv.1506.04967

Matuschek, H., Kliegl, R., Vasishth, S., Baayen, H. and Bates, D. (2017). Balancing Type I error and power in linear mixed models. *Journal of Memory and Language*, 94, 305-315. doi:10.1016/j.jml.2017.01.001

Examples

```
spec <- build_spec(list(name = "d", seed = 1, design_kind = "within",
  include_items = TRUE, n_subject = 10, n_item = 8, factor_name = "cond",
  lev1 = "a", lev2 = "b", intercept = 6, effect = 0.05,
  subj_int_sd = 0.12, subj_slope_sd = 0.04, subj_corr = 0.2,
  item_int_sd = 0.08, item_slope_sd = 0.02, item_corr = -0.1,
  family = "shifted_lognormal", resp_name = "", sigma = 0.3, shift = 200))
model_formula(spec)
model_formula(spec, "zcp")
model_formula(spec, "intercepts")
```

pilotr_example *Example design specifications shipped with pilotr*

Description

pilotr ships eight ready-to-run design specifications, as JSON, in the package's `examples/` directory. The 'Python' package and the no-code app use the same files, so a design written for one runs unchanged in the others. `pilotr_example()` lists them, or returns the path to one for `load_spec()`.

Usage

```
pilotr_example(name = NULL)
```

Arguments

`name`	The base name of an example, with or without the `.json` extension, for example `"between_2group_gaussian"`. When `NULL` (the default), the available example names are returned in place of a path.

Value

When `name` is `NULL`, a character vector of the available example names. Otherwise, the full path to that example's JSON file, ready to pass to `load_spec()`.

See Also

`load_spec()` to read a specification and `simulate_design()` to simulate from it.

Examples

```
pilotr_example()                       # the available examples
spec <- load_spec(pilotr_example("between_2group_gaussian"))
head(simulate_design(spec))
```

power_curve_mixed *Power curve over sample size for a mixed-effects design*

Description

Sweep the number of subjects and compute mixed-effects power at each. Pass the result to `target_n()` for the sample size at which power crosses a target, with an interval on it. It calls `sweep_spec()` over `units$subject$n`. Requires the `lme4` and `lmerTest` packages.

Usage

```
power_curve_mixed(
  spec,
  subject_ns,
  focal = NULL,
  n_sims = 60,
  alpha = 0.05,
  workers = 1,
  seeds = c("independent", "shared"),
  alternative = c("two.sided", "greater", "less")
)
```

Arguments

spec	A design specification (path or list).
subject_ns	A numeric vector of subject counts to evaluate.
focal	The fixed effects to test, or the named sets of coefficients to test jointly, as in power_mixed().
n_sims	Number of Monte Carlo replicates per point. A power estimate carries a Monte Carlo standard error of about sqrt(p * (1 - p) / n_sims), reported alongside it as power_mcse. The default of 60 gives a standard error of 0.065 at a power of 0.5, which is too coarse to support a claim about a design. Raise it to at least 200 for planning.
alpha	Significance level of the test, two-sided unless alternative says otherwise.
workers	Number of local worker processes over which to spread the replicates at each grid point. The default of 1 runs serially, and any worker count returns results identical to a serial run. One pool of that many processes serves the whole sweep.
seeds	How the sample sizes are seeded: "independent", the default, or "shared", as in sweep_spec()'s .seeds.
alternative	The alternative hypothesis of every test, "two.sided", "greater" or "less", as in power_mixed().

Details

Like power_mixed(), this runs pilotr's own simulation loop over the portable design specification. It differs from simr (Green and MacLeod, 2016) and mixedpower (Kumle, Võ and Draschkow, 2021) in the same two ways: the specification drives it and it reports Type S and Type M errors. With workers > 1, one worker pool is created once and reused across all sample sizes.

Each sample size draws its own replicates, from the seed sweep_spec() gives its grid point, so the points of the curve are independent, as target_n() assumes when it fits them. seeds = "shared" runs every point from the specification's own seed instead, and the points then share their noise (see sweep_spec()).

For any axis other than sample size, call sweep_spec() directly. Effect size is the axis a design analysis most often needs after sample size, and design_conditions() builds the coefficient overrides for it.

The curve is the input to `target_n()`, which is where the sample size a preregistration quotes should come from. Reading the crossing off a plot instead judges points whose Monte Carlo intervals overlap, and reports the answer without the interval that goes with it.

In a crossed design, the number of items limits how far adding subjects can raise power. When the effect of a condition varies from item to item, as a by-item random slope says it does, part of the standard error of the effect comes from the items sampled. That part shrinks only as items are added. As subjects are added, the standard error therefore falls towards a floor set by the number of items. Power at a fixed number of items levels off below 1 (Westfall, Kenny and Judd, 2014; Judd, Westfall and Kenny, 2017). The probit curve that `target_n()` fits rises towards 1 and cannot bend into such a plateau. A curve that levels off near the target can then be solved far from the true crossing. One that levels off just below the target never crosses it, but Monte Carlo noise can lift a simulated power over the target, and the curve can then be solved as though it crossed. To compare designs that trade subjects for items, sweep `units$item$n` with `sweep_spec()` at each of a few subject counts and solve each curve for the number of items, as the power vignette shows. Power over items levels off in the same way, at a ceiling set by the by-subject slope and the number of subjects. The subject counts must be large enough to put that ceiling well above the target.

Value

A data frame with one row per sample size and focal effect, with columns `n_subject`, `effect`, `true`, `power`, `power_mcse`, `power_lo`, `power_hi`, `n_significant`, `type_s`, `type_s_lo`, `type_s_hi`, `type_m`, `type_m_mcse`, the `n_attempted`, `n_returned`, `n_converged`, `n_singular` and `n_warning` fit counts, and `fitter`, as in `power_mixed()`. `fitter` names the direction of a one-sided test. With sets of coefficients in `focal`, `effect` holds each set's name, a column `coefficients` after it lists the set's coefficients, and `fitter` names the joint test. `n_singular` typically falls as the sample size rises, so reading it down the sweep shows where the model becomes supportable.

References

Green, P. and MacLeod, C. J. (2016). SIMR: An R package for power analysis of generalized linear mixed models by simulation. *Methods in Ecology and Evolution*, 7(4), 493-498. doi:10.1111/2041-210x.12504

Judd, C. M., Westfall, J. and Kenny, D. A. (2017). Experiments with more than one random factor: Designs, analytic models, and statistical power. *Annual Review of Psychology*, 68, 601-625. doi:10.1146/annurevpsych122414033702

Kumle, L., Võ, M. L.-H. and Draschkow, D. (2021). Estimating power in (generalized) linear mixed models: An open introduction and tutorial in R. *Behavior Research Methods*, 53(6), 2528-2543. doi:10.3758/s13428021015460

Westfall, J., Kenny, D. A. and Judd, C. M. (2014). Statistical power and optimal design in experiments in which samples of participants respond to samples of stimuli. *Journal of Experimental Psychology: General*, 143(5), 2020-2045. doi:10.1037/xge0000014

See Also

`target_n()` to solve the returned curve for a sample size, `sweep_spec()` for any other axis and `design_conditions()` for effect-size grids.

Examples

```
if (requireNamespace("lme4", quietly = TRUE) &&
    requireNamespace("lmerTest", quietly = TRUE)) {
  spec <- build_spec(list(name = "p", seed = 1, design_kind = "within",
    include_items = TRUE, n_subject = 12, n_item = 12, factor_name = "cond",
    lev1 = "a", lev2 = "b", intercept = 6, effect = 0.05,
    subj_int_sd = 0.12, subj_slope_sd = 0.04, subj_corr = 0.2,
    item_int_sd = 0.08, item_slope_sd = 0.02, item_corr = -0.1,
    family = "shifted_lognormal", resp_name = "", sigma = 0.3, shift = 200))
  # n_sims is small so the example runs quickly. Use 200 or more for real planning.
  power_curve_mixed(spec, subject_ns = c(12, 18), n_sims = 8)
}
```

`power_design` *Simulation-based power and design analysis for a two-group Gaussian design*

Description

Estimate power by repeatedly simulating from the specification and applying a two-sample t-test, alongside the Type S (sign) and Type M (magnitude) design-analysis errors of Gelman and Carlin (2014).

Usage

```
power_design(
  spec,
  n_sims = 1000,
  alpha = 0.05,
  workers = 1,
  cl = NULL,
  alternative = c("two.sided", "greater", "less")
)
```

Arguments

`spec`	A design specification (path or list) for a two-group Gaussian design with one row per subject (see Details).
`n_sims`	Number of Monte Carlo replicates. A power estimate carries a Monte Carlo standard error of about `sqrt(p * (1 - p) / n_sims)`, reported as `power_mcse`, and `type_s` and `type_m` average over the significant replicates alone, so they settle more slowly still. At least 200 replicates are advisable for study planning.
`alpha`	Significance level of the test, two-sided unless `alternative` says otherwise.
`workers`	Number of local worker processes over which to spread the replicates. The default of 1 runs serially. Because every replicate takes its own seed from `replicate_seeds()`, any worker count returns results identical to a serial run.

cl	A cluster made by parallel::makeCluster() to run the replicates on, in place of one started for this call, in which case workers is ignored. The cluster is left running for its owner to stop, and through sweep_spec() one cluster serves every grid point.
alternative	The test's alternative hypothesis: "two.sided", the default, or "greater" or "less" for a one-sided test of a directional prediction (see Details).

Details

The t-test treats every row as an independent observation, which is valid only when each subject contributes one row and no rows share a cluster. power_design() therefore takes a Gaussian design with exactly one two-level factor between subjects and no item unit, within factor or grouping factor besides subject. Continuous predictors and by-subject random effects are allowed, since with one row per subject they vary independently from row to row. Any other design is refused with a message naming what makes its rows correlated. power_mixed() fits the model that such a design implies. Take 30 subjects crossed with 20 items, with by-subject and residual standard deviations of 1 and a by-item one of 0.3. With no true effect, a t-test of every row is significant in about half of all replicates.

Each level of the factor needs at least two subjects, so a design needs at least 4. The subjects are split into two blocks in level order, and the second level holds floor(N / 2). With one subject at a level, the pooled variance rests on the other level alone, and with one at each it has no degree of freedom at all.

A specification with no coefficient for the factor's contrast has a true effect of 0, so type_s and type_m are NaN.

Power, Type S and Type M are each estimated from a finite number of replicates, so each is reported with its Monte Carlo error. Power has its standard error and a 95% Wilson interval, one of the intervals Brown, Cai and DasGupta (2001) recommend in place of the Wald interval. It stays between 0 and 1 and does not shrink to a point when power is 0 or 1. Type S and Type M rest on the significant replicates alone, often a handful. Type S has a Wilson interval, because its standard error is zero whenever no sign is wrong. Type M has its standard error, the standard deviation of the exaggeration ratios over the square root of their number.

A directional prediction calls for a one-sided test, which alternative selects. The estimate is the mean of the second level minus that of the first, whose true value is true_effect. "greater" tests whether it is positive and "less" whether it is negative, by stats::t.test() with the same alternative. The one-sided p-value is half the two-sided one when the estimate lies in the tested direction. A one-sided test at alpha therefore rejects in exactly the replicates in which the two-sided test at 2 * alpha rejects with the estimate in that direction. Type S and Type M are taken over the replicates significant in the tested direction. When that is the direction of the true effect, Type S is 0 by construction, and Type M is the exaggeration among the replicates that would support the prediction.

A one-sided power at alpha, in the direction of the true effect, can be read off a two-sided result at 2 * alpha: it is power * (1 - type_s). Type M is recovered only when that result's Type S is 0, since the wrong-signed replicates cannot otherwise be told apart from the others.

Value

A list with elements n_sims, alpha, alternative, power, power_mcse (its Monte Carlo standard error), power_lo and power_hi (its 95% Wilson interval), n_significant, true_effect and

mean_estimate. Then come type_s (sign-error rate among significant replicates), type_s_lo and type_s_hi (its 95% Wilson interval), type_m (mean exaggeration ratio among significant replicates) and type_m_mcse (its Monte Carlo standard error). Both design-analysis quantities are NaN when no replicate reached significance and when the true effect is zero, as in the null condition design_conditions() produces: neither is defined without a true value to compare against, and Type M divides by it. Their interval and standard error are NaN then too, and type_m_mcse is also NaN when a single replicate reached significance, since a standard deviation needs two.

References

Brown, L. D., Cai, T. T. and DasGupta, A. (2001). Interval estimation for a binomial proportion. *Statistical Science*, 16(2), 101-133. doi:10.1214/ss/1009213286

Gelman, A. and Carlin, J. (2014). Beyond power calculations: Assessing Type S (sign) and Type M (magnitude) errors. *Perspectives on Psychological Science*, 9(6), 641-651. doi:10.1177/1745691614551642

Examples

```
spec <- build_spec(list(name = "d", seed = 1, design_kind = "between",
  factor_name = "group", lev1 = "a", lev2 = "b", n_subject = 64,
  intercept = 100, effect = 5, family = "gaussian", resp_name = "", sigma = 10))
# n_sims is small so the example runs quickly. Use 200 or more for real planning.
power_design(spec, n_sims = 50)
```

---

power_mixed *Simulation-based power and design analysis for a mixed-effects design*

Description

For each replicate, simulate from the ground-truth specification, fit the model the specification implies and test each focal fixed effect. The fitter follows the response family: lmerTest::lmer() with Satterthwaite p-values for the continuous families, and lme4::glmer() with a Wald z for accuracy and count responses (see Details). Reports power together with the Type S and Type M errors of Gelman and Carlin (2014). Requires the lme4 and lmerTest packages.

Usage

```
power_mixed(
  spec,
  focal = NULL,
  formula = NULL,
  prep = NULL,
  n_sims = 100,
  alpha = 0.05,
  workers = 1,
  cl = NULL,
```

```
  alternative = c("two.sided", "greater", "less"),
  fallback = c("singular", "warning")
)
```

Arguments

spec	A design specification (path or list).
focal	The fixed effects to test. NULL, the default, tests every coefficient in the specification and takes the true values from it. A character vector names the effects and leaves the true values unknown, which suppresses Type S and Type M. A named numeric vector gives both, which is how to test against a value other than the one simulated. Interaction effects follow the model's column naming, so a specification key a:b is the focal name a_b. A named list of character vectors names sets of coefficients, each tested jointly, as in list(interaction = c("A1_B", "A2_B")) (see Details).
formula	Optional lme4 formula. If NULL, it is derived from the specification by model_formula(), and the response family chooses the fitter. A formula given here is fitted as a linear model whatever the family (see Details). A ladder of models simulates a simplification rule: a character vector of model_formula() structures, such as c("maximal", "zcp", "intercepts"), fitted as the derived model is, or a list of formulas, fitted as written. A rung takes its name from the ladder, or else from its structure or its position.
prep	Optional function mapping a simulated data set to the modelling data. If NULL, it is derived by model_data().
n_sims	Number of Monte Carlo replicates. A power estimate carries a Monte Carlo standard error of about sqrt(p * (1 - p) / n_sims), and type_s and type_m average over the significant replicates alone, so they settle more slowly still. At least 200 replicates are advisable for study planning.
alpha	Significance level of the test, two-sided unless alternative says otherwise.
workers	Number of local worker processes over which to spread the replicates. The default of 1 runs serially. Because the replicate seeds are derived once from the specification's seed, any worker count returns results identical to a serial run. The mixed-model fits dominate the cost, so the speed-up is close to linear in the number of cores.
cl	A cluster made by parallel::makeCluster() to run the replicates on, in place of one started for this call, in which case workers is ignored. The cluster is left running for its owner to stop. Through sweep_spec(), as in sweep_spec(spec, "fixed$coefficients$cond", values, power_mixed, cl = cl), one cluster serves every grid point.
alternative	The alternative hypothesis of every focal effect's test: "two.sided", the default, or "greater" or "less" for a one-sided test of a directional prediction (see Details).
fallback	When a replicate moves on to the next rung of a ladder: "singular", the default, after a fit that is boundary-singular or failed, and "warning" also after a fit with a warning or an optimiser convergence message. It has no effect on a single formula.

Details

`power_mixed()` runs pilotr's own simulation loop over the portable design specification. `simr` (Green and MacLeod, 2016) and `mixedpower` (Kumle, Võ and Draschkow, 2021) offered such simulations first, and pilotr is indebted to them. It differs from them in two ways: the portable cross-language specification drives it and it reports the Type S and Type M design-analysis errors beside power.

The analysis model comes from the specification rather than from this function: the formula comes from `model_formula()` and the data from `model_data()`, so the analysis matches the process that generated the data. Both can be given directly, to fit a deliberately different analysis model, as when checking how a misspecified model behaves.

The fitter follows the response family, so that the estimates are on the scale of the specification's coefficients. A `gaussian`, `lognormal`, `shifted_lognormal` or `exgaussian` response is fitted by `lmerTest::lmer()`, the two lognormal families on the log scale to which `model_data()` takes them, and each effect is tested with Satterthwaite's degrees of freedom. A `bernoulli` or `poisson` response is fitted by `lme4::glmer()` with the binomial or Poisson family, on the logit or log scale, and each effect is tested with a Wald z. A model with no random terms, such as that of a between-subjects design with one row per subject, is fitted by `stats::lm()` or `stats::glm()` in the same way. The result's `fitter` names the fitter and the test. A Wald z treats the estimate over its standard error as normal and so ignores the uncertainty in the variance components. Bolker et al. (2009) recommend it for the fixed effects of a GLMM without overdispersion, which these models are, but it rejects too often when the clusters are few. In binary GLMMs with fewer than 30 clusters, a test with as many degrees of freedom as observations, close to a z test, rejected more often than its nominal level (Li and Redden, 2015). The power of a `bernoulli` or `poisson` design with few subjects or items is therefore somewhat overstated.

pilotr has no frequentist model on the logit scale for an `ordinal` or `beta` response. Such a response is fitted by the linear model on its own scale, while the specification's coefficients are on the logit scale. The function therefore warns, and withholds the mean estimate and Type M (`NA`), since both would compare the two scales. Type S compares signs alone, and the link keeps the sign of a main effect unless an interaction reverses that effect between the levels of another factor. It does not keep the sign of an interaction. A bounded response compresses differences near the ends of its range, so an interaction that is positive on the logit scale can be negative on the response scale. Type S is therefore withheld for an interaction as well. Power is the rate at which the linear model's test rejects. `generate_design_analysis()` writes a Bayesian design analysis on the link scale, with the model `brms_bridge()` derives.

A `formula` given here is fitted as written, by `lmerTest::lmer()`, or by `stats::lm()` when it has no random terms, whatever the family, since only its author knows the scale of its response. With such a formula, or a `prep` that changes the response, and `focal = NULL`, the true values still come from the specification and are on its scale, which need not be the model's. A named numeric `focal` gives the true values on the model's scale.

An analysis plan may instead preregister a simplification rule: fit the maximal model, and when the fit is singular or fails to converge, drop the correlations, then the slopes (Matuschek et al., 2017). Barr et al. (2013) recommend the maximal model and suggest dropping its correlations when it fails to converge. `formula` given as a ladder of models simulates such a rule. Each replicate fits the rungs in turn and keeps the first fit that `fallback` accepts, or the last rung's fit when it accepts none. `c("maximal", "zcp", "intercepts")` names structures that `model_formula()` builds, which are fitted as the default model is, with the fitter the family chooses. A list of formulas gives rungs that

are fitted as written. Power, Type S and Type M are then those of the rule. `rung_shares` gives the share of the replicates analysed at each rung, and the fit counts describe the fit each replicate kept. A rule that can end on random intercepts alone ignores, in the replicates that end there, how much the effect varies over subjects and items. A test that ignores that variation rejects a true null hypothesis too often (Barr et al., 2013). Run the null condition of `design_conditions()` through the same ladder to see how often, as the power vignette does. A single formula is unaffected by `fallback`.

Every reported rate carries its Monte Carlo standard error and a Wilson interval, because a proportion over a finite number of replicates is an estimate rather than a fact. At the default 100 replicates, a power near 0.5 has a standard error of 0.05. Type S and Type M average over the significant replicates alone, which may be few. Each carries its own Monte Carlo error. Type S has a Wilson interval, since its standard error would be zero whenever no sign is wrong. Type M has its standard error, the standard deviation of the exaggeration ratios over the square root of their number.

A directional prediction calls for a one-sided test, which `alternative` selects for every focal effect. `"greater"` tests whether the coefficient is positive and `"less"` whether it is negative. The one-sided p-value is the upper or lower tail of the statistic behind the two-sided one. That is Satterthwaite's t, the t of `stats::lm()` with its residual degrees of freedom, or a Wald z referred to the normal distribution. The one-sided p-value is half the two-sided one when the estimate lies in the tested direction. A one-sided test at `alpha` therefore rejects in exactly the replicates in which the two-sided test at `2 * alpha` rejects with the estimate in that direction. Type S and Type M are taken over the replicates significant in the tested direction. When that is the direction of the true effect, Type S is 0 by construction, and Type M is the exaggeration among the replicates that would support the prediction. `fitter` names the direction, as in `"lmerTest::lmer, Satterthwaite t, one-sided (greater)"`.

A one-sided power at `alpha`, in the direction of the true effect, can be read off a two-sided result at `2 * alpha`: it is `power * (1 - type_s)`. Type M is recovered only when that result's Type S is 0, since the wrong-signed replicates cannot otherwise be told apart from the others.

A factor with more than two levels enters the model as several contrast columns, and its interaction with another factor as several product columns. The hypothesis a study preregisters about such an effect is then often the joint test of those coefficients, and its power differs from that of any one of them. A named list as `focal`, such as `list(interaction = c("A1_B", "A2_B"))` for the interaction of a three-level factor with a two-level one, plans that test. Each set is tested against the hypothesis that all its coefficients are zero. A linear mixed model has Satterthwaite's F from `lmerTest::contest()` (Kuznetsova et al., 2017). A `glmer()` or `glm()` fit has the Wald chi-square of the set's estimates and their covariance matrix, and an `lm()` fit has the F test that compares the model with and without the set. A set of one coefficient therefore gives the p-value of the single-coefficient test, with F equal to t squared or a chi-square equal to z squared. Under a ladder of models, each set is tested on the fit the simplification rule kept. `fitter` names the joint test, as in `"lmerTest::lmer, Satterthwaite F"`. A joint test has no direction, so `alternative` must stay `"two.sided"`. Nor has it a single estimate, so the true value, the mean estimate, Type S and Type M are `NA`. A likelihood-ratio test is not offered, since it would refit every replicate by maximum likelihood and is anti-conservative for fixed effects in small samples (Pinheiro and Bates, 2000).

Value

An object of class `pilotr_power`, a list whose per-run elements are `n_sims`, `alpha`, `fitter`, `n_attempted`, `n_returned`, `n_converged`, `n_singular` and `n_warning`, and whose per-effect elements are vectors named by focal effect: `power`, `power_mcse`, `power_lo`, `power_hi`, `n_significant`,

true_effect, mean_estimate, type_s, type_s_lo, type_s_hi, type_m and type_m_mcse. With a single focal effect, each of those has length one, so result$power is a single number. fitter names the fitter and its test, such as "lme4::glmer (binomial), Wald z", with the direction of a one-sided test added, as in "lme4::glmer (binomial), Wald z, one-sided (greater)". Under a one-sided test, n_significant and the design-analysis quantities count the replicates significant in the tested direction. mean_estimate and type_m are NA for an ordinal or beta response fitted on its own scale, and so is the type_s of an interaction (see Details). type_s_lo and type_s_hi, the Wilson interval of type_s, and type_m_mcse, the Monte Carlo standard error of type_m, are NA wherever their rate is, and type_m_mcse also when a single replicate reached significance.

A list focal adds a per-run element sets, the sets as given, and names the per-effect elements by set. Each set's true_effect, mean_estimate and Type S and Type M quantities are then NA, and fitter names the joint test.

power is the proportion of significant results among the replicates that returned an estimate for that effect, not among n_sims. The counts report the fit outcomes separately, because a fit can return a usable estimate while still being boundary-singular or carrying a convergence warning. n_returned counts replicates that yielded a fit, n_converged those that did so with neither a warning nor a singular fit, n_singular those where lme4::isSingular() was true and n_warning those with a warning or optimiser convergence message other than lme4's singular-fit notice, which n_singular already counts. Singular and warning fits are retained in power, since their fixed-effect estimates remain interpretable and discarding them would bias the result: singularity is not independent of the variance estimates that produce it. A large n_singular means the model being fitted is richer than the design can support at that sample size, which is common in crossed designs (Bates et al., 2015; Matuschek et al., 2017), and is worth reporting alongside the power.

A ladder adds two per-run elements: rung_shares, the share of the replicates analysed at each rung, named by rung and summing to 1, and fallback, the rule applied. A replicate in which no rung's fit met the rule counts at the last rung. The fit counts then describe the fit each replicate kept, so n_singular counts the replicates left with a singular fit at the last rung.

References

Barr, D. J., Levy, R., Scheepers, C. and Tily, H. J. (2013). Random effects structure for confirmatory hypothesis testing: Keep it maximal. *Journal of Memory and Language*, 68(3), 255-278. doi:10.1016/j.jml.2012.11.001

Gelman, A. and Carlin, J. (2014). Beyond power calculations: Assessing Type S (sign) and Type M (magnitude) errors. *Perspectives on Psychological Science*, 9(6), 641-651. doi:10.1177/1745691614551642

Green, P. and MacLeod, C. J. (2016). SIMR: An R package for power analysis of generalized linear mixed models by simulation. *Methods in Ecology and Evolution*, 7(4), 493-498. doi:10.1111/2041-210x.12504

Kumle, L., Võ, M. L.-H. and Draschkow, D. (2021). Estimating power in (generalized) linear mixed models: An open introduction and tutorial in R. *Behavior Research Methods*, 53(6), 2528-2543. doi:10.3758/s13428021015460

Kuznetsova, A., Brockhoff, P. B. and Christensen, R. H. B. (2017). lmerTest package: Tests in linear mixed effects models. *Journal of Statistical Software*, 82(13), 1-26. doi:10.18637/jss.v082.i13

Pinheiro, J. C. and Bates, D. M. (2000). *Mixed-effects models in S and S-PLUS*. Springer. doi:10.1007/b98882

Bates, D., Kliegl, R., Vasishth, S. and Baayen, H. (2015). Parsimonious mixed models. *arXiv*. doi:10.48550/arXiv.1506.04967

Matuschek, H., Kliegl, R., Vasishth, S., Baayen, H. and Bates, D. (2017). Balancing Type I error and power in linear mixed models. *Journal of Memory and Language*, 94, 305-315. doi:10.1016/j.jml.2017.01.001

Bolker, B. M., Brooks, M. E., Clark, C. J., Geange, S. W., Poulsen, J. R., Stevens, M. H. H. and White, J.-S. S. (2009). Generalized linear mixed models: A practical guide for ecology and evolution. *Trends in Ecology & Evolution*, 24(3), 127-135. doi:10.1016/j.tree.2008.10.008

Li, P. and Redden, D. T. (2015). Comparing denominator degrees of freedom approximations for the generalized linear mixed model in analyzing binary outcome in small sample cluster-randomized trials. *BMC Medical Research Methodology*, 15, 38. doi:10.1186/s128740150026x

See Also

`precision_design()` for the interval-width and ROPE analogue, and `sweep_spec()` to run this over a grid of sample sizes or effect sizes.

Examples

```
if (requireNamespace("lme4", quietly = TRUE) &&
    requireNamespace("lmerTest", quietly = TRUE)) {
  spec <- build_spec(list(name = "p", seed = 1, design_kind = "within",
    include_items = TRUE, n_subject = 12, n_item = 12, factor_name = "cond",
    lev1 = "a", lev2 = "b", intercept = 6, effect = 0.05,
    subj_int_sd = 0.12, subj_slope_sd = 0.04, subj_corr = 0.2,
    item_int_sd = 0.08, item_slope_sd = 0.02, item_corr = -0.1,
    family = "shifted_lognormal", resp_name = "", sigma = 0.3, shift = 200))
  # n_sims is small so the example runs quickly. Use 200 or more for real planning.
  power_mixed(spec, n_sims = 10)
}
```

`power_report` *Write a sample-size justification from a power result*

Description

Turns the result of `power_mixed()`, `power_design()` or `precision_design()`, with the `target_n()` result that solved a curve for the sample size, into a sample-size justification in Markdown, ready for a preregistration, a registered report or a grant application.

Usage

```
power_report(
  x,
  solved = NULL,
  spec_file = NULL,
```

```
  justification = NULL,
  null_result = NULL,
  rope = NULL
)

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

Arguments

x	For power_report(), the result of power_mixed(), power_design() or precision_design() at one design. A curve goes to target_n(), and its result to solved. For print(), a report.
solved	Optional result of target_n(), or of solve_curve() for an axis other than sample size, which gives the desired power and the value solved for it.
spec_file	Optional path of the specification file the analysis read. The report gives the MD5 checksum of its bytes, its seed and its units.
justification	Optional justification of each focal effect's size: one string for every focal effect, or a character vector named by focal effect. An effect without one is reported as having none.
null_result	Optional result of the same analysis with every focal effect set to zero. A result whose focal effects are not zero is refused.
rope	The region of practical equivalence given to precision_design(), as a half-width or an interval, which the report states. It applies to a precision result alone.
...	Ignored, present for consistency with the generic.

Details

The sections follow the recommendations of Lakens (2022, Table 4) for reporting an a-priori power analysis. The primary analysis comes first, with its model, test, alpha and sidedness, then the desired power and the sample size solved for it, with its interval. Each focal effect follows, with its true value, the justification given for that value and its power. Last comes the Type I error rate of the test, its rejection rate when every focal effect is zero. For a true null hypothesis, Table 4 asks instead for the power of a test for the absence of an effect, such as an equivalence test. The simulation is then accounted for as Morris, White and Crowther (2019) recommend for simulation studies: the replicates simulated and those that returned an estimate, the fits that were singular or carried a warning and the Monte Carlo standard error of every rate. The report ends with what a reader needs to reproduce the run: the software and its versions, the specification's seed and an MD5 checksum of the specification file's bytes. The versions are those loaded when the report is written, so write it in the session that ran the analysis.

Each rate's Monte Carlo standard error and 95% Wilson interval are computed in the report from the number of significant replicates and the number of replicates behind the rate, with the functions that compute them in the results. Type S has its Wilson interval from the number of wrong signs among the significant replicates. The 'Python' package's power_report() writes the same text, byte for byte, from the same counts.

The rates are those of the design `x` simulated. A sample size solved from a curve need not be that design's, so a justification usually pairs the solved size with a run of `power_mixed()` at that size. `spec_file` names the specification of that run, and the report lists its units. A result carries neither its specification nor its seed, so without `spec_file` the report says that both are missing.

`null_result` is the result of the same analysis of a design whose focal effects are all zero, such as the null condition that `design_conditions()` puts first. Its rejection rate is the Type I error rate of the test, which for a Wald z with few subjects or items can exceed alpha, and which a preregistered simplification rule can raise (see `power_mixed()`).

A `precision_design()` result gives a report on the interval and the region of practical equivalence in place of the test. Each focal effect then has its mean interval width, the probability of a meaningful effect, with the interval wholly outside the region, and the probability of practical equivalence, with the interval wholly inside it. `rope` states the region, since the result does not record it. A `target_n()` result solved over `p_meaningful` gives the desired probability of a meaningful decision. Only the R package has this section, as only it has `precision_design()`.

Value

A character string of class `pilotr_report` holding the Markdown, which ends in a newline. Printing it shows the text, and `cat(report, file = "justification.md")` saves it. The same arguments always give the same text.

References

Lakens, D. (2022). Sample size justification. *Collabra: Psychology*, 8(1), 33267. doi:10.1525/collabra.33267

Morris, T. P., White, I. R. and Crowther, M. J. (2019). Using simulation studies to evaluate statistical methods. *Statistics in Medicine*, 38(11), 2074-2102. doi:10.1002/sim.8086

See Also

`target_n()` for the sample size a curve implies, and `design_conditions()` for the null condition.

Examples

```
spec <- build_spec(list(name = "d", seed = 1, design_kind = "between",
  factor_name = "group", lev1 = "a", lev2 = "b", n_subject = 64,
  intercept = 100, effect = 5, family = "gaussian", resp_name = "", sigma = 10))
# n_sims is small so the example runs quickly. Use 200 or more for real planning.
res <- power_design(spec, n_sims = 50)
spec_file <- tempfile(fileext = ".json")
writeLines(spec_json(spec), spec_file)
power_report(res, spec_file = spec_file,
  justification = "Half a standard deviation, the smallest difference worth acting on.")
```

precision_curve *Precision and ROPE curve over sample size*

Description

Sweep the number of subjects and report the ROPE decision probabilities at each size. Pass the result to `solve_curve()` for the minimum analysable N at which a focal effect reaches a determinate decision with a target probability, such as 0.90, together with an interval on it. Calls `precision_design()` and so requires the `lme4` package.

Usage

```
precision_curve(
  spec,
  focal = NULL,
  subject_ns,
  formula = NULL,
  prep = NULL,
  rope = 0.05,
  n_sims = 60,
  workers = 1,
  seeds = c("independent", "shared"),
  fallback = c("singular", "warning")
)
```

Arguments

`spec`	A design specification (path or list).
`focal`	The focal effects, as in `precision_design()`. `NULL` uses every coefficient in the specification.
`subject_ns`	A numeric vector of subject counts to evaluate.
`formula`	Optional `lme4` formula. If `NULL`, it is derived by `model_formula()`, and the response family chooses the fitter, as in `precision_design()`. A ladder of models, as in `precision_design()`, simulates a simplification rule at every sample size.
`prep`	Optional data-preparation function. If `NULL`, it is derived by `model_data()`.
`rope`	The region of practical equivalence, as in `precision_design()`: a non-negative half-width `r` for the region from `-r` to `r`, or a pair `c(lower, upper)` with `lower < upper`. Keep the effects worth detecting clearly outside the region, because the probability of a determinate meaningful decision about an effect inside it cannot rise above 0.5 however large the sample, so the curve would fall with N rather than rise.
`n_sims`	Number of Monte Carlo replicates per sample size. `p_meaningful` and `p_equivalent` are proportions over the replicates that produced an estimate, so each is reported with its Monte Carlo standard error and Wilson interval. The default of 60 gives

a standard error of 0.065 at a rate of 0.5, which is too coarse to support a claim about a design. Raise it to at least 200 for planning.

`workers` Number of local worker processes over which to spread the replicates at each sample size. The default of 1 runs serially, and any worker count returns results identical to a serial run. One pool of that many processes serves the whole sweep.

`seeds` How the sample sizes are seeded: `"independent"`, the default, or `"shared"`, as in `sweep_spec()`'s `.seeds`.

`fallback` The rule of a ladder, `"singular"` or `"warning"`, as in `precision_design()`.

Details

It calls `sweep_spec()` over `units$subject$n`. For any other axis, call `sweep_spec()` directly. Effect size is the axis a design analysis most often needs next, and `design_conditions()` builds the coefficient overrides for it.

Each sample size draws its own replicates, from the seed `sweep_spec()` gives its grid point, so the points of the curve are independent, as `solve_curve()` assumes when it fits them. `seeds = "shared"` runs every point from the specification's own seed instead, and the points then share their noise (see `sweep_spec()`). With `workers > 1`, one worker pool is created and reused across all sample sizes.

Reading the crossing off the returned curve, or off a plot of it, is what `solve_curve()` replaces. At the replicate counts these runs are usually given, neighbouring points on the curve are not significantly different from one another, so a sample size judged by eye is a point estimate with an unstated and often wide uncertainty behind it.

Value

A data frame with one row per focal effect and sample size. It has the columns returned by `precision_design()`, including the Monte Carlo standard errors, the Wilson interval bounds and the `n_returned`, `n_converged`, `n_singular` and `n_warning` fit counts, and an `n_subject` column. For a ladder, it also has the share of each rung at each sample size.

See Also

`solve_curve()` to solve the returned curve for a target decision probability, `sweep_spec()` for any other axis and `design_conditions()` for effect-size grids.

Examples

```
if (requireNamespace("lme4", quietly = TRUE)) {
  spec <- build_spec(list(name = "pr", seed = 1, design_kind = "within",
    include_items = TRUE, n_subject = 12, n_item = 12, factor_name = "cond",
    lev1 = "a", lev2 = "b", intercept = 6, effect = 0.05,
    subj_int_sd = 0.12, subj_slope_sd = 0.04, subj_corr = 0.2,
    item_int_sd = 0.08, item_slope_sd = 0.02, item_corr = -0.1,
    family = "shifted_lognormal", resp_name = "", sigma = 0.3, shift = 200))
  # n_sims is small so the example runs quickly. Use 200 or more for real planning.
  precision_curve(spec, focal = c(effect = 0.05), subject_ns = c(12, 18), rope = 0.02,
                  n_sims = 8)
```

```
}
```

`precision_design` *Precision and ROPE design analysis at a fixed sample size*

Description

A fast frequentist analogue of a Bayesian highest-density-interval-versus-ROPE design analysis. Across Monte Carlo replicates, fit the model and record, for each focal fixed effect, whether its 95% confidence interval falls entirely outside a region of practical equivalence (a practically meaningful effect) or entirely inside it (practical equivalence), along with the expected interval width. Requires the `lme4` package.

Usage

```
precision_design(
  spec,
  focal = NULL,
  formula = NULL,
  prep = NULL,
  rope = 0.05,
  n_sims = 100,
  workers = 1,
  cl = NULL,
  fallback = c("singular", "warning")
)
```

Arguments

`spec`	A design specification (path or list).
`focal`	The focal effects. `NULL`, the default, analyses every coefficient in the specification and takes the true values from it. A named numeric vector maps coefficient names to their true values, and a character vector names them without their true values. Interaction effects follow the model's column naming, so a specification key `a:b` is the focal name `a_b`. A list of coefficient sets, which `power_mixed()` tests jointly, is refused, since a set has no single estimate to build an interval around.
`formula`	Optional `lme4` formula. If `NULL`, it is derived from the specification by `model_formula()`, and the response family chooses the fitter. A formula given here is fitted as a linear model whatever the family (see Details). A ladder of models simulates a simplification rule: a character vector of `model_formula()` structures, such as `c("maximal", "zcp", "intercepts")`, fitted as the derived model is, or a list of formulas, fitted as written, as in `power_mixed()`.

prep — Optional function mapping a simulated data set to the modelling data. If NULL, it is derived by model_data(), which adds the analysis response .y, on the log scale for the lognormal families, and builds the contrast and interaction columns, so focal names follow the auto-formula (interactions written as a_b).

rope — The region of practical equivalence, as a half-width or as an interval. A single non-negative number r is the region from -r to r, so that an effect with abs(beta) < r is treated as practically equivalent to zero. A pair c(lower, upper) with lower < upper is the region between those bounds, which need not be centred on zero, as for a prediction that an effect lies within a stated interval. Any other value is refused. Keep the effects worth detecting clearly outside the region, because the probability of a determinate meaningful decision about an effect inside it cannot rise above 0.5 however large the sample.

n_sims — Number of Monte Carlo replicates. p_meaningful and p_equivalent are proportions over the replicates that returned an estimate (n_returned), so they carry a Monte Carlo standard error of about sqrt(p * (1 - p) / n_sims) and move in coarse steps when n_sims is small. At least 200 replicates are advisable for real planning.

workers — Number of local worker processes over which to spread the replicates. The default of 1 runs serially. Because every replicate takes its own seed from replicate_seeds(), any worker count returns results identical to a serial run.

cl — A cluster made by parallel::makeCluster() to run the replicates on, in place of one started for this call, in which case workers is ignored. The cluster is left running for its owner to stop, and through sweep_spec() one cluster serves every grid point.

fallback — When a replicate moves on to the next rung of a ladder: "singular", the default, after a fit that is boundary-singular or failed, and "warning" also after a fit with a warning or an optimiser convergence message. It has no effect on a single formula.

Details

The interval is a Wald approximation: the estimate plus or minus 1.96 standard errors from the model's variance-covariance matrix. This fixed-z interval is chosen for speed and for comparability across replicates. In small samples, it is somewhat narrower than a Satterthwaite t interval, so p_meaningful and mean_ci_width are slightly optimistic at small sample sizes.

The fitter follows the response family, as in power_mixed(). Since no p-value is needed, a gaussian, lognormal, shifted_lognormal or exgaussian response is fitted by lme4::lmer(). A bernoulli or poisson response is fitted by lme4::glmer() with the binomial or Poisson family, and a model with no random terms by stats::lm() or stats::glm(). The interval is therefore on the scale of the specification's coefficients, and so is the region of practical equivalence that rope sets. That is the log scale for the lognormal families and poisson, and the logit scale for bernoulli. A glmer() fit has the Wald interval that Bolker et al. (2009) recommend for a GLMM without overdispersion. With few subjects or items, it is likewise narrower than it should be, since the test it inverts rejects too often when the clusters are few (Li and Redden, 2015). The fitter column names the fitter.

pilotr has no frequentist model on the logit scale for an ordinal or beta response. Such a response is fitted by the linear model on its own scale, where a region of practical equivalence set on the logit

scale of the coefficients means nothing. The function warns, and withholds the decision probabilities with their Monte Carlo standard errors and bounds (NA). mean_ci_width stays, as the width of an interval on the response scale. generate_design_analysis() applies an interval-and-ROPE rule on the link scale, in a Bayesian model.

A formula given here is fitted as written, by lme4::lmer(), or by stats::lm() when it has no random terms, whatever the family. With such a formula, or a prep that changes the response, and focal = NULL, the true values still come from the specification and are on its scale, which need not be the model's. A named numeric focal gives the true values on the model's scale.

formula given as a ladder of models simulates a preregistered simplification rule, as in power_mixed(). Each replicate fits the rungs in turn and keeps the first fit that fallback accepts, or the last rung's fit when it accepts none. The intervals and decisions are then those of the rule, and one column per rung gives the share of the replicates analysed at it.

Value

A data frame with one row per focal effect and columns param, true, mean_ci_width, p_meaningful, p_meaningful_mcse, p_meaningful_lo, p_meaningful_hi, p_equivalent, p_equivalent_mcse, p_equivalent_lo, p_equivalent_hi, n_attempted, n_returned, n_converged, n_singular, n_warning and fitter. Each decision proportion is reported with its Monte Carlo standard error (*_mcse) and Wilson interval bounds (*_lo, *_hi), because a proportion over a finite number of replicates is an estimate rather than a fact. The interval behind mean_ci_width and the ROPE decisions is the Wald approximation described in Details. fitter names the fitter and the interval, such as "lme4::glmer (binomial), Wald z". For an ordinal or beta response fitted on its own scale, the decision columns are NA (see Details).

The decision proportions are taken over n_returned, the replicates that produced an estimate. The remaining counts separate the fit outcomes, because a fit can return a usable estimate while still being boundary-singular or carrying a convergence warning: n_converged counts replicates with neither, n_singular those where lme4::isSingular() was true and n_warning those with a warning or optimiser convergence message other than lme4's singular-fit notice, which n_singular already counts. Singular and warning fits are retained, since their fixed-effect estimates remain interpretable and discarding them would bias the result. A large n_singular means the model being fitted is richer than the design can support at that sample size, which is common in crossed designs (Bates et al., 2015; Matuschek et al., 2017).

A ladder adds one column per rung, rung_<name>, holding the share of the replicates analysed at that rung, the same on every row. The shares sum to 1, and a replicate in which no rung's fit met the rule counts at the last rung. The fit counts then describe the fit each replicate kept.

References

Bates, D., Kliegl, R., Vasishth, S. and Baayen, H. (2015). Parsimonious mixed models. *arXiv*. doi:10.48550/arXiv.1506.04967

Matuschek, H., Kliegl, R., Vasishth, S., Baayen, H. and Bates, D. (2017). Balancing Type I error and power in linear mixed models. *Journal of Memory and Language*, 94, 305-315. doi:10.1016/j.jml.2017.01.001

Bolker, B. M., Brooks, M. E., Clark, C. J., Geange, S. W., Poulsen, J. R., Stevens, M. H. H. and White, J.-S. S. (2009). Generalized linear mixed models: A practical guide for ecology and evolution. *Trends in Ecology & Evolution*, 24(3), 127-135. doi:10.1016/j.tree.2008.10.008

Li, P. and Redden, D. T. (2015). Comparing denominator degrees of freedom approximations for the generalized linear mixed model in analyzing binary outcome in small sample cluster-randomized trials. *BMC Medical Research Methodology*, 15, 38. doi:10.1186/s128740150026x

Examples

```
if (requireNamespace("lme4", quietly = TRUE)) {
  spec <- build_spec(list(name = "pr", seed = 1, design_kind = "within",
    include_items = TRUE, n_subject = 12, n_item = 12, factor_name = "cond",
    lev1 = "a", lev2 = "b", intercept = 6, effect = 0.05,
    subj_int_sd = 0.12, subj_slope_sd = 0.04, subj_corr = 0.2,
    item_int_sd = 0.08, item_slope_sd = 0.02, item_corr = -0.1,
    family = "shifted_lognormal", resp_name = "", sigma = 0.3, shift = 200))
  # n_sims is small so the example runs quickly. Use 200 or more for real planning.
  precision_design(spec, focal = c(effect = 0.05), rope = 0.02, n_sims = 10)
}
```

---

print.pilotr_bridge *Print a brms bridge*

Description

Writes the ready-to-fit `brms` model held in the object's `code` element, which is the form the bridge is meant to be read in and the one to copy into a script. The other elements (`formula`, `family`, `priors`) are the same model in parts, for a caller assembling its own code, and are left to `str()` or to `$`.

Usage

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

Arguments

`x`	A `pilotr_bridge` object, as returned by `brms_bridge()`.
`...`	Ignored, present for consistency with the generic.

Value

`x`, invisibly.

Examples

```
spec <- build_spec(list(name = "d", seed = 1, design_kind = "between",
  factor_name = "g", lev1 = "a", lev2 = "b", n_subject = 20,
  intercept = 0, effect = 0.4, family = "gaussian", resp_name = "y", sigma = 1))
print(brms_bridge(spec))
```

print.pilotr_power *Print a simulation-based power result*

Description

Shows the fit accounting, with the share of the replicates analysed at each rung when the result comes from a ladder, and then two tables with one row per focal effect. The first gives each power estimate beside its Monte Carlo standard error and Wilson interval, so that the precision of the estimate is as visible as the estimate. The second gives Type S beside its Wilson interval and Type M beside its Monte Carlo standard error, both taken over the significant replicates alone.

Usage

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

Arguments

x	A pilotr_power object, as returned by power_mixed().
digits	Number of decimal places for the reported rates and for the interval bounds.
...	Ignored, present for consistency with the generic.

Details

A result whose focal named sets of coefficients lists the coefficients of each set before the first table, which has one row per set and no true value. It has no second table, since a joint test has no single estimate from which Type S or Type M could be taken.

Value

x, invisibly.

replicate_seeds *Seeds for the replicates of a Monte Carlo run*

Description

The seeds pilotr's power and precision loops give to their replicates, derived from a specification's own seed. Exported so that a hand-written replicate loop, or a cluster array task that has to reproduce one replicate on its own, can use the same rule and so land on the same data.

Usage

```
replicate_seeds(base, n)
```

Arguments

base	The specification's seed.
n	How many replicate seeds to return.

Details

The simpler rule, `base + (i - 1)`, will not do, because consecutive seeds are not independent streams in this generator. Seeding is affine in the seed, setting `s1` to `1 + (seed mod 2147483562)` and `s2` from `s1`, and each component recurrence is linear, so at every position `k` the draws of consecutive seeds differ by very nearly the same amount mod 1. For the first draw, that step is -0.0091: seeds 1000 to 1005 begin with 0.7027, 0.6937, 0.6846, 0.6755, 0.6665 and 0.6574. Draws in arithmetic progression mod 1 with step `d` correlate `1 - 6|d|(1 - |d|)`, so over 2,000 replicates the first draw of replicate `i` correlated 0.95 with the first draw of replicate `i + 1`. A longer warm-up changes `d` and leaves the progression in place: with 50 warm-up draws in place of 10, the step is -0.32 and the correlation -0.30.

An arithmetic scramble does not fix that. Adding a Weyl increment and applying a Lehmer step leaves the seeds in arithmetic progression with a longer stride, and since the seeding rule is affine in the seed, the first draws remained correlated at -0.27. Any affine rule for the seeds keeps the draws in progression, so no linear change to the rule removes it. Matsumoto et al. (2007) describe the same defect in the initialisation of other linear generators.

Drawing the seeds from the shared generator does work. The combined generator is designed so that its successive outputs behave as independent draws, and the seeds inherit that: the same measurement gives -0.02, and a Ljung-Box test over the resulting replicate means moves from p below 0.0001 to p of 0.94. The whole vector costs `n` draws, computed once for the loop. Duplicates are skipped, so no two replicates are handed the same seed and silently produce identical data. The skipping is deterministic, so the R and 'Python' implementations still agree.

The first replicate's seed is drawn like the rest, so it is not the specification's own seed.

Value

A numeric vector of `n` distinct seeds.

References

Matsumoto, M., Wada, I., Kuramoto, A. and Ashihara, H. (2007). Common defects in initialization of pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation*, 17(4), 15. doi:10.1145/1276927.1276928

Examples

```
replicate_seeds(90210, 5)

# The same data as replicate 3 of a power run over this specification.
spec <- build_spec(list(name = "d", seed = 90210, design_kind = "between",
  factor_name = "g", lev1 = "a", lev2 = "b", n_subject = 20,
  intercept = 0, effect = 0.5, family = "gaussian", resp_name = "", sigma = 1))
spec$seed <- replicate_seeds(90210, 3)[3]
head(simulate_design(spec), 3)
```

`response_variance` *Variance components of the linear predictor*

Description

Decompose the variance of a design's linear predictor into the part contributed by the fixed effects, the part contributed by each grouping factor's random effects and the residual variance added by the response family. Useful for putting a region of practical equivalence or a smallest effect size of interest on a known scale, and for seeing which term dominates a design before committing to it.

Usage

```
response_variance(spec)
```

Arguments

`spec`	A design specification (path or list).

Details

The fixed component is the variance, across rows, of the linear predictor with every random-effect standard deviation set to zero. It is read off the design the specification actually produces, so it needs no assumption about how a between-unit factor divides the units or about whether the predictors are independent.

Each grouping factor's component is the average over rows of `x' Sigma x`, where `Sigma` is the covariance the specification asks for and `x` collects that row's values of the intercept and each random-slope column. This is exact for the realised design: it averages over the random-effect distribution analytically. Drawing from that distribution and estimating a variance from the drawn effects of, say, 30 subjects would carry a sampling error of about a quarter of the component itself, far too much to calibrate a region of practical equivalence against.

The cost is one simulation for the fixed part plus one per distinct random-slope column, which for a typical design is a handful.

Every component is on the scale of the linear predictor, which is what makes them comparable. For `lognormal` and `shifted_lognormal`, that is the log of the response, which is also the scale the auto-derived analysis model works on. For `bernoulli`, `poisson`, `ordinal` and `beta`, it is the latent scale behind the link, so the residual is the distribution-specific variance used to compute the intraclass correlation and R-squared of a generalised mixed model (Nakagawa, Johnson and Schielzeth, 2017).

Three of those four are derived from the process pilotr simulates and are exact for it. A `bernoulli` row is drawn as `1[u < invlogit(eta)]`, equivalently `logit(u) < eta`, so the latent error is a standard logistic variate of variance `pi^2 / 3`, and `ordinal` compares the same uniform against cumulative thresholds and inherits it. For `beta`, `logit(Y)` has variance `trigamma(a) + trigamma(b)` exactly.

`poisson` is the one approximation, because a count of zero has no logarithm and a log-scale variance cannot be measured directly. The trigamma form is used, following the recommendation for the log link. It agrees closely with the alternatives once the mean exceeds about 5 and diverges sharply

below it, where a log-scale variance is barely meaningful: at a mean of 0.5 it gives 4.93 against 1.10 for the lognormal approximation. Read the Poisson residual as an order of magnitude when counts are rare.

The total includes the variance the fixed effects contribute. It is the denominator of an R-squared, and of an effect or a region of practical equivalence stated in standard deviations of the outcome, but it is not the denominator of Cohen's d. For designs in which participants respond to stimuli, Westfall, Kenny and Judd (2014) define d as the difference between two condition means divided by the standard deviation within a condition. The square of that standard deviation sums the random intercept variances, the residual and the random slope variances, each slope variance weighted by the squared contrast code. The fixed effects are left out (see also Brysbaert and Stevens, 2018). In a balanced design with codes that sum to zero, that is `total - fixed`. Covariates belong to the fixed part, so a d defined this way is conditional on them.

A published d therefore converts to a coefficient as `b = d * sqrt(total - fixed) / (c2 - c1)`, where `c2 - c1` is the difference between the two contrast codes. For codes of -0.5 and 0.5, that difference is 1. The d is on the scale of the linear predictor, so for the families with a link it is a latent-scale d. `calibrate_response()` with `denominator = "random"` rescales a design so that `total - fixed` is 1, after which the coefficient of a contrast coded -0.5 and 0.5 is d.

Value

A named list of variance components: `fixed`, one entry per grouping factor, `residual` and `total` (their sum).

References

Brysbaert, M. and Stevens, M. (2018). Power analysis and effect size in mixed effects models: A tutorial. *Journal of Cognition*, 1(1), 9. doi:10.5334/joc.10

Nakagawa, S., Johnson, P. C. D. and Schielzeth, H. (2017). The coefficient of determination R2 and intra-class correlation coefficient from generalized linear mixed-effects models revisited and expanded. *Journal of the Royal Society Interface*, 14(134), 20170213. doi:10.1098/rsif.2017.0213

Westfall, J., Kenny, D. A. and Judd, C. M. (2014). Statistical power and optimal design in experiments in which samples of participants respond to samples of stimuli. *Journal of Experimental Psychology: General*, 143(5), 2020-2045. doi:10.1037/xge0000014

See Also

`calibrate_response()` to rescale a design to a target total variance.

Examples

```
spec <- pilotr_example("crossed_mixed_rt")
response_variance(spec)

# The coefficient that gives the related-unrelated contrast, coded -0.5 and 0.5, a published d
# of 0.2 on the log scale of this design.
v <- response_variance(spec)
0.2 * sqrt(v$total - v$fixed) / (0.5 - (-0.5))

# Every family reports a residual, including those whose outcome is discrete, so the components
```

```
# are a complete decomposition and their ratios read as the design's intraclass correlations.
v <- response_variance(pilotr_example("ordinal_likert_between"))
round(1 - v$residual / v$total, 3)   # share of latent variance that is structural
```

---

run_app *Launch the pilotr no-code app*

Description

Runs the bundled Shiny app locally. When the package is installed, the app calls the package's functions directly. Running locally gives each user a private R process, so no one else's work waits on theirs.

Usage

```
run_app(..., async = NULL)
```

Arguments

`...`	Further arguments passed to `shiny::runApp()`, such as `port` and `launch.browser`.
`async`	If TRUE (the default when 'future' and 'promises' are installed), the app runs each power analysis and power curve in a background R process. That leaves the app's own R process free for other sessions, such as a second browser tab, while the session that started the run waits for it. A parallel future plan you have set is used as it stands. Over a sequential plan, the default, `run_app()` sets a multisession plan of two workers while the app runs. Either way, the plan in force before the call is in force again once the app stops. With FALSE, power runs in the app's own R process and no plan is set.

Details

The app needs the 'shiny' and 'ggplot2' packages, which pilotr only suggests, and its Verify button needs the 'callr' package.

Each power analysis and power curve in the app runs in one R process, and the session that asked for it waits for the result. Its controls and outputs do not update until the run ends. For a heavy run, download the specification and call `power_design()` or `power_mixed()` from the console, where `workers` spreads the replicates over your local cores.

Value

No return value. It is called for its side effect of launching the Shiny application, which blocks the R session until the app is closed.

Examples

```
## Not run:
run_app()

## End(Not run)
```

`simulate_design` *Simulate a data set from a design specification*

Description

Generate an analysis-ready data set from a portable design specification: a linear predictor built from fixed effect sizes (categorical contrasts, continuous predictors and their interactions) plus crossed by-subject and by-item random intercepts and slopes, mapped through the chosen response family.

Usage

```
simulate_design(spec, validate = TRUE)
```

Arguments

`spec`	A design specification, given either as a path to a JSON file or as an already-parsed list (for example from `build_spec()` or `load_spec()`).
`validate`	Whether to validate the specification first, via `validate_spec()`. The default `TRUE` catches the errors that would otherwise pass silently, such as a mistyped coefficient key, which resolves to no column and so sets that effect to zero. Validation costs a few milliseconds, so the replicate loops behind the power and precision functions validate once and then pass `FALSE`. There is rarely a reason to set it directly. Any other value validates leniently, as in `load_spec()`.

Value

A data frame with one row per observation, containing a `subject` column, an optional `item` column, any grouping, factor and continuous-predictor columns and the response column named by the specification. A design with a counterbalanced factor also has a `list` column after `item`, holding the list each subject follows, counted from 1.

Examples

```
spec <- build_spec(list(name = "demo", seed = 1, design_kind = "between",
  factor_name = "group", lev1 = "control", lev2 = "treatment", n_subject = 40,
  intercept = 100, effect = 5, family = "gaussian", resp_name = "", sigma = 10))
head(simulate_design(spec))
```

`solve_curve` *Solve a simulated design curve for the value that meets a target*

Description

Take the curve a sweep has already produced, fit the decision rate against the swept value and solve for the value at which the rate meets a target. The solved value comes with a confidence interval, because a value read off a simulated curve without one hides the uncertainty of the simulation.

Usage

```
solve_curve(
  curve,
  target,
  x = NULL,
  y = NULL,
  n = NULL,
  effect = NULL,
  transform = "sqrt",
  level = 0.95
)
```

Arguments

`curve`	A curve, as returned by `power_curve_mixed()`, `precision_curve()` or `sweep_spec()`: one row per swept value, with the swept value, a decision rate and the number of replicates behind it.
`target`	The decision rate to solve for, strictly between 0 and 1.
`x`	Name of the column holding the swept value. `NULL`, the default, takes the leading column, which holds a grid index when the sweep was over lists.
`y`	Name of the column holding the decision rate. `NULL`, the default, takes `power` or `p_meaningful`, whichever is present.
`n`	The number of replicates behind each rate, either the name of a column or a numeric value. `NULL`, the default, takes `n_returned`, `n_converged` or `n_sims`, whichever is present.
`effect`	Which focal effect to solve for, when the curve holds more than one. Matched against the `effect` or `param` column. `NULL`, the default, uses every row, which is correct only when the curve holds one effect.
`transform`	The scale the swept value is fitted on: `"sqrt"` (the default, for a sample size), `"identity"` or `"log"`.
`level`	Confidence level for the reported interval.

Details

The input is the data frame `power_curve_mixed()`, `precision_curve()` or `sweep_spec()` returns, used as it stands. The swept value is taken from the leading column, which is where all three put it, and the rate from `power` or `p_meaningful`, whichever the curve carries. Each rate is a proportion over a known number of replicates, and that count, read from `n_returned`, `n_converged` or `n_sims`, weights the fit: a rate over 200 replicates should count for more than a rate over 20.

The fit is a binomial regression with a probit link, and the solved value is the swept value at which the fitted rate equals `target`. The probit is chosen because power is a normal tail probability: under the normal approximation to a two-group comparison, the probit of power is linear in the square root of the sample size, so the model has the shape a design analysis already implies. Measured against `stats::power.t.test()` across twelve combinations of effect size and target power on each of three grid shapes, at 400 replicates a point, the solved sample size fell within 2.6% of the analytic answer on average. A logit fitted the same way did as well on that draw, at 2.5%, and worse on another, at 3.5% against 2.2%. Curves this noisy do not separate the two links, and the probit is chosen for its shape.

The interval is the delta-method interval of `MASS::dose.p()`, computed on the scale named by `transform` and mapped back, so it is symmetric on that scale and asymmetric on the natural one. That asymmetry is correct: at the top of a power curve, a given change in rate costs far more sample size than the same change lower down. In the checks against `stats::power.t.test()` above, Fieller's interval for a ratio (Fieller, 1954), tried as the alternative, covered the analytic answer as often, at nearly the same width, so the interval `MASS::dose.p()` reports is kept. Where the two-parameter model does not describe the curve, the interval is widened by the heterogeneity factor of probit analysis, Pearson's chi-square over its degrees of freedom (Finney, 1971), reported as `dispersion`. It is floored at 1, so a well-fitting curve is left alone and a badly fitting one cannot report a narrower interval than its own residuals justify.

The default `transform` of `"sqrt"` suits a sample-size axis, where a rate rises with the square root of the sample size. Sweep something else, an effect size or a random-effect standard deviation, and `"identity"` is usually right.

A sweep over the conditions `design_conditions()` builds is a sweep over lists, so its leading column holds a grid index, and solving it as it stands would solve for that index. Either name the column holding the effect with `x`, such as `true` in the rows of `power_mixed()`, or sweep one coefficient by its own path. The curve from `sweep_spec(spec, "fixed$coefficients$cond", c(0, 0.02, 0.04), power_mixed)` holds the other coefficients at their values and leads with a column named `cond`. From that curve, `solve_curve(curve, target = 0.8, effect = "cond", transform = "identity")` returns the minimum detectable effect.

Nothing here extrapolates. A curve whose rates do not straddle the target is refused, with the range it did cover reported, and so is a fit that solves outside the swept range. A curve whose fitted slope cannot be told from zero is refused too: the crossing is then compatible with any value at all, and an interval that said otherwise would be false.

A curve over subjects in a crossed design need not rise towards 1. When an effect varies from item to item, as a by-item random slope says it does, power at a fixed number of items levels off below 1 (Westfall, Kenny and Judd, 2014; Judd, Westfall and Kenny, 2017). Adding subjects cannot lift it past that ceiling. The probit model rises towards 1 and cannot follow such a plateau. A curve that levels off near the target can then be solved far from the true crossing. One that levels off just below the target never crosses it, but Monte Carlo noise can lift a simulated rate over the target, and the curve can then be solved as though it crossed. To compare designs that trade subjects for items,

solve curves over the number of items, one at each of a few subject counts, from `sweep_spec()` over `units$item$n`. Power over items levels off in the same way, at a ceiling set by the by-subject slope and the number of subjects. The subject counts must be large enough to put that ceiling well above the target.

What the interval covers is the Monte Carlo uncertainty of the fit, not the gap between the fitted shape and the true curve, and it treats the points of the curve as independent. By default, every sweep in pilotr draws each point's replicates from a seed of its own (see `sweep_spec()`). Over 1,000 simulated curves for each combination of four effect sizes and three target powers, at 400 replicates a point, the nominal 95% interval covered the analytic sample size in 96.5%, 95.9% and 96.2% of solves on the three grid shapes. For an effect size swept at a fixed sample size, it covered the analytic effect in 95.7% of solves. With `seeds = "shared"` (`.seeds` in `sweep_spec()`), every point reused the specification's seed, and the figures were 87.8%, 94.0%, 93.5% and 70.3%. A curve assembled by hand, from separate runs that each started from the same seed, shares its noise in the same way. In the calibration's 36 checks, the solved size sat within 2.6% of the analytic answer on average and within 7.7% at worst. Nearly all of that error is the Monte Carlo noise the interval is describing, and it falls with the square root of the replicate count. Raise the count far enough and the interval narrows onto a fitted shape that is still slightly the wrong shape, so replicates alone do not make a solved size arbitrarily accurate. The remedies are more replicates, a finer grid or a design with a closed form to check against. A finer grid helps only while its points are independent. With shared seeds, its points share more of their noise, and the tight grid, whose points lie closest together, covered 87.8% where the coarse one covered 94.0%.

Value

A list with elements `value` (the solved swept value), `lo` and `hi` (its confidence bounds), `level`, `target`, `se` (the delta-method standard error on the fitted scale, the scale on which the interval is symmetric), `dispersion` (the heterogeneity factor applied, 1 where the model fits), `x` and `y` (the columns used), `transform`, `intercept` and `slope` (the fitted coefficients), `n_points` (the number of curve points the fit used) and `x_min` and `x_max` (the swept range). A bound is allowed to fall outside that range. When one does, the sweep was too narrow to pin the value down and should be widened. A `dispersion` well above 1 says the curve is not the shape the model assumes, so the solve needs a wider grid or more replicates before it can be trusted.

References

Fieller, E. C. (1954). Some problems in interval estimation. *Journal of the Royal Statistical Society: Series B*, 16(2), 175-185. doi:10.1111/j.25176161.1954.tb00159.x

Finney, D. J. (1971). *Probit analysis* (3rd ed.). Cambridge University Press.

Judd, C. M., Westfall, J. and Kenny, D. A. (2017). Experiments with more than one random factor: Designs, analytic models, and statistical power. *Annual Review of Psychology*, 68, 601-625. doi:10.1146/annurevpsych122414033702

Westfall, J., Kenny, D. A. and Judd, C. M. (2014). Statistical power and optimal design in experiments in which samples of participants respond to samples of stimuli. *Journal of Experimental Psychology: General*, 143(5), 2020-2045. doi:10.1037/xge0000014

See Also

`target_n()` for the sample-size case, and `power_curve_mixed()`, `precision_curve()` and `sweep_spec()`

for the curves this consumes.

Examples

```
spec <- build_spec(list(name = "s", seed = 1, design_kind = "between", n_subject = 40,
  factor_name = "group", lev1 = "a", lev2 = "b", intercept = 0, effect = 0.7,
  family = "gaussian", resp_name = "score", sigma = 1))
# n_sims is small so the example runs quickly. Use 200 or more for real planning.
curve <- sweep_spec(spec, "units$subject$n", c(20, 40, 60, 80), power_design, n_sims = 50)
solved <- solve_curve(curve, target = 0.8)
unlist(solved[c("value", "lo", "hi")])

# This design has an analytic answer to check against, in total subjects across the two
# groups. It falls inside the interval, which fifty replicates a point make a wide one.
2 * stats::power.t.test(delta = 0.7, sd = 1, power = 0.8)$n
```

`spec_from_model` *Build a design specification from a fitted mixed model*

Description

Read a design specification off a linear mixed model already fitted with `lme4::lmer()` or `lmerTest::lmer()`. The fixed effects, the random-effect standard deviations and correlations, and the residual standard deviation are taken from the fit, and the numbers of subjects and items may be raised at the same time, so that a pilot study or a published model becomes the starting point of a power analysis. The fixed effects of a pilot are its point estimates, and a study powered for them tends to be underpowered. A message names them on every call, and `sesoi` or `effects = "safeguard"` replaces them (see Details). Requires the `lme4` package.

Usage

```
spec_from_model(
  fit,
  name = NULL,
  seed = 1,
  n_subject = NULL,
  n_item = NULL,
  family = NULL,
  round = NULL,
  effects = c("estimate", "safeguard"),
  safeguard_level = 0.6,
  sesoi = NULL,
  counterbalance = TRUE
)
```

Arguments

fit	A fitted linear mixed model, of class lmerMod (from lme4::lmer()) or lmerModLmerTest (from lmerTest::lmer()). Anything else is refused with a message saying what to pass instead.
name	A label for the returned specification. Defaults to "from_model".
seed	The master seed of the returned specification. Defaults to 1.
n_subject	Number of subjects for the returned specification. Defaults to the number of levels the fit had, and is normally raised above it, since a specification is read off a pilot fit to plan a larger study.
n_item	Number of items, treated the same way as n_subject. An error when the model has no second crossed grouping factor to act as items, because there is then no item unit to resize.
family	The response family of the returned specification. NULL, the default, gives "gaussian", which is what a lmer fit is on the scale it was fitted on. A single string names another family, and a list such as list(family = "shifted_lognormal", shift = 200) also supplies the parameters of that family which a Gaussian fit cannot provide.
round	Decimal places for the simulated response, passed through to response.round. NULL, the default, leaves the response unrounded.
effects	Where the coefficients of the returned specification come from. "estimate", the default, takes the fit's point estimates. "safeguard" takes the bound of each coefficient's two-sided safeguard_level interval nearer zero, and stops with an error when that interval includes zero (see Details). Either way, a coefficient named in sesoi takes the value given there, and the intercept is the fit's.
safeguard_level	The confidence level of the intervals behind effects = "safeguard", a single number greater than 0 and less than 1. The default of 0.6 follows Perugini, Gallucci and Costantini (2014).
sesoi	NULL, the default, or a named numeric vector that sets coefficients to a smallest effect size of interest on the scale of the fit, as in sesoi = c(cond = 0.03). The names are the returned specification's coefficient keys, which write an interaction as a:b, and a name that is not among them is refused.
counterbalance	Whether a factor that the pilot counterbalanced sets counterbalance in the returned specification, which then declares specification version 0.4 (see Details). TRUE, the default, plans the pilot's design, unless its subjects each saw only some of the items or its model has a term named list. FALSE returns a fully crossed specification that releases reading version 0.3 accept, with a warning that the power computed from it is too high.

Details

Choosing plausible random-effect standard deviations with nothing to read them from is the step that most often stops a simulation-based power analysis before it starts. The tutorials on the method agree on the answer, which is to take the variance components from a pilot fit or from a published model (Green and MacLeod, 2016; Kumle et al., 2021; DeBruine and Barr, 2021). This function

takes them from such a fit, and the specification it returns can be enlarged and passed straight to `simulate_design()` or `power_mixed()`.

The fixed effects are another matter. They are the pilot's point estimates, which a small pilot measures imprecisely, and a pilot followed up because its effect looked promising has on average overestimated that effect. A study powered for such an estimate therefore tends to have less power than planned (Albers and Lakens, 2018). The variance components are the part of a pilot worth carrying over. The effect is better set to a smallest effect size of interest (SESOI), the smallest effect of theoretical or practical importance. For an effect keyed `cond`, as in the `crossed_mixed_rt` example, pass `sesoi = c(cond = 0.03)`, or set `spec$fixed$coefficients$cond <- 0.03` on the returned specification. Every call reports in a message which coefficients it took from the fit, with their values.

When no SESOI can be argued for, `effects = "safeguard"` takes safeguard values (Perugini, Gallucci and Costantini, 2014). Each coefficient not named in `sesoi` becomes the bound of its two-sided `safeguard_level` interval nearer zero, by default that of the 60% interval. The interval is the estimate plus or minus a quantile times the standard error from `vcov(fit)`. For an `lme4` fit, the quantile is normal, `qnorm((1 + safeguard_level) / 2)`, which is 0.84 at the default. The normal quantile treats the standard error as known, so the interval is too narrow when the subjects or items are few. For a `lmerTest` fit, the quantile is a t quantile on the Satterthwaite degrees of freedom that `lmerTest` reports, which allow for the uncertainty in the variance components, although only approximately. When `lmerTest` cannot give them for every coefficient, all coefficients take the normal quantile, and the message names the distribution used. An interval that includes zero stops the call with an error naming the effect. The pilot then leaves even the sign of the effect open, and a value of zero would plan a study of no effect at all. Such an effect needs a value in `sesoi`. By default, the conditions that `design_conditions()` builds for `sweep_spec()` include one with every named effect at zero, which shows how often a design declares an effect that is not there.

Whatever their source, the coefficients of the specification are the effects its data are simulated with. A named numeric `focal` in `power_mixed()` changes only the true values against which Type S and Type M are computed. The simulated effect stays as it was, so powering a study for another effect means changing the specification.

Every number is taken on the scale the model was fitted on, and the returned family is Gaussian by default, because that is what a `lmer` fit is. A model of log reaction times therefore yields an intercept, coefficients and residual standard deviation on the log scale, which is the right thing for the `lognormal` and `shifted_lognormal` families, whose linear predictor is on that scale as well. Ask for one of those with `family`, in the list form when the family needs a parameter the fit cannot supply, as in `family = list(family = "shifted_lognormal", shift = 200)`.

Whether a numeric column of the model frame was a contrast-coded factor or a continuous covariate is not recorded anywhere in a fitted model, so it is inferred here. A column that is a factor, a character vector or a logical vector is treated as categorical and its own contrast coding is read with `stats::contrasts()`, which makes the emitted contrast-column names agree with the coefficient names lme4 produced. A numeric column with exactly two distinct values is also treated as categorical, on the reasoning that a two-valued numeric predictor in a factorial experiment is a coded factor far more often than it is a covariate. Any other numeric column becomes a continuous predictor, with `mean` and `sd` taken from the data, one value per unit for a unit-level variable so that an unbalanced design does not distort them. This is a heuristic, and a message reports it on every call. A genuine two-valued covariate has to be moved from `factors` to `predictors` by hand.

A column is placed as `between` a unit when it holds one value within every member of that unit, and as `vary_within` when it takes several values inside every unit. The same test gives a continuous

predictor its varies_by, which becomes "observation" when the predictor varies inside every unit. A model that codes one factor twice, once as a factor for its fixed effect and once as a numeric contrast for a random slope, gives two separate specification terms, because the two columns are separate columns in the model frame and nothing in the fit ties them together.

Two or more factors placed between the same unit become one factor whose levels are their cells, and a message says so. The first-listed factor varies slowest, and each label joins the components' labels with ".". Two between factors over one unit need specification version 0.4, which allocates them jointly. Up to version 0.3, each was assigned to blocks of its unit on its own, which confounded their effects, and validate_spec() refuses them in a specification declaring an earlier version. The factor of cells allocates the units as version 0.4 allocates the separate factors, and every release that reads version 0.3 reads it, so the returned specification declares 0.3. The combined factor carries every component's contrast columns under the names the fit gave them, so each coefficient the fit estimated, interactions included, still names a column. Every cell then receives units, in equal numbers whenever the number of units is a multiple of the number of cells.

Interactions need one further step. lme4 writes the interaction of two model-frame columns as a:b, which is already the specification's convention, but model_data() gives an interaction its own product column named a_b, so a specification built from a fit of pilotr's own modelling data would otherwise acquire a spurious independent term. Such a column is recognised by checking the product identity in the data, which both avoids mistaking a column called z_freq for an interaction and settles where to split a name with several underscores. A recognised product column is reported and re-keyed to a:b, and it contributes no factor or predictor of its own.

Grouping factors named subject and item are used as they stand. Otherwise, the one with the most levels becomes subject, the largest remaining factor that is crossed with it becomes item, and every other grouping factor becomes an extra random entry with the over and n fields that pilotr's additional grouping factors take, over being decided by which unit the factor partitions. Any renaming is reported by a message and recorded in the group_mapping attribute of the result. When each subject saw only some of the items, the item unit gains a per_subject count, so that the recovered design keeps the partial crossing of the original.

A factor that varies within both units is crossed with every subject-item pair unless it sets counterbalance, a field of specification version 0.4. A pilot run in counterbalanced lists, as priming studies usually are, showed each subject each item at one level only. The levels rotated across lists of subjects, so that every item appeared at each level. When every subject-item pair of the pilot holds one level of a factor, and every subject and every item holds two or more, the returned specification sets counterbalance to TRUE on that factor and declares version 0.4. Each subject then meets each item at one level, the levels rotating over subjects and items as described under Counterbalanced lists in the specification (spec/SPEC.md). The simulated data name each subject's list in a column list. Factors rotated together, as two two-level factors over four lists, are all counterbalanced, and a message names them. Releases that read no version above 0.3 refuse such a specification.

counterbalance = FALSE returns the crossed specification instead, with a warning, and releases that read no version above 0.3 accept it. It has twice the pilot's observations per subject for a two-level factor, three times for a three-level one and four times for two two-level factors rotated together. The power computed from it is therefore too high. Take the variance components of the crossed_mixed_rt example, with 30 subjects, 24 items and an effect of 0.05 on the log scale. By the formulas of Judd, Westfall and Kenny (2017), power is then about 0.80 for the crossed design and 0.55 for the counterbalanced one. The counterbalanced study needs about 1.8 times the subjects to match the crossed one. A release that reads no version above 0.3 takes a two-level design written with a list factor between subjects and an item-set factor between items, as in the two-list example

among the worked encodings of the specification. The crossed specification and the warning also come back, whatever `counterbalance` says, when each subject of the pilot saw only some of the items. The specification then keeps `per_subject`, which version 0.4 does not combine with `counterbalance`, since that field needs every subject to see every item. They come back as well when the pilot's model has a column or grouping factor named `list`, as a model that fits the list beside the condition does. Version 0.4 gives that name to the column holding each subject's list, and no other column may share it. The warning then asks for the term to be renamed in the pilot's data and the model refitted.

Alternatively, keep the crossed specification and drop the rows that a counterbalanced study would not observe, through the `prep` argument of `power_mixed()` and `precision_design()`. That function receives each simulated data set and returns the modelling data, so it ends by calling `model_data()`. With `workers > 1`, it runs in worker processes that have not attached pilotr and cannot see the caller's variables, so it has to call `pilotr::model_data()` and carry the specification with it. A function made by another function carries it once that function has forced its argument, as in this rotation over two lists for the `crossed_mixed_rt` example, whose factor is `condition`:

```
counterbalance <- function(spec) {
  force(spec)
  function(d) {
    level <- match(d$condition, spec$factors[[1]]$levels) - 1L
   pilotr::model_data(spec, d[(d$subject %% 2L + d$item + level) %% 2L == 0L, ])
  }
}
power_mixed(spec, prep = counterbalance(spec), workers = 4)
```

A boundary-singular pilot fit is carried across as it stands, with a variance estimated at zero or a correlation estimated at exactly plus or minus one. Those values read the fit faithfully. They also show that the random-effect structure was richer than the pilot could support, and a power analysis built on them inherits that (Bates et al., 2015). To avoid it, enlarge the pilot design, or simplify the structure before refitting.

Value

A design specification as a nested list, carrying the lowest `spec_version` that reads it, 0.4 when it sets `counterbalance` and 0.3 otherwise, and validated with `validate_spec()`, so it can be passed directly to `simulate_design()`, `power_mixed()` or `spec_json()`. Two attributes record the readings that the fit did not settle on its own: `group_mapping`, a named character vector giving the specification unit each of the fit's grouping factors became, and `column_kinds`, a named character vector giving each model-frame column the classification `"factor"` or `"predictor"`. A third, `effects`, is a list recording where the coefficients came from. Its `method` is the value of `effects`, its `level` the `safeguard_level` applied (`NA` with `"estimate"`) and its `sesoi` the keys of the coefficients set from `sesoi`. All three are attributes rather than fields so that the specification itself stays within the portable schema.

References

Albers, C. and Lakens, D. (2018). When power analyses based on pilot data are biased: Inaccurate effect size estimators and follow-up bias. *Journal of Experimental Social Psychology*, 74, 187-195. doi:10.1016/j.jesp.2017.09.004

Bates, D., Kliegl, R., Vasishth, S. and Baayen, H. (2015). Parsimonious mixed models. *arXiv*. doi:10.48550/arXiv.1506.04967

DeBruine, L. M. and Barr, D. J. (2021). Understanding mixed-effects models through data simulation. *Advances in Methods and Practices in Psychological Science*, 4(1). doi:10.1177/2515245920965119

Green, P. and MacLeod, C. J. (2016). SIMR: An R package for power analysis of generalized linear mixed models by simulation. *Methods in Ecology and Evolution*, 7(4), 493-498. doi:10.1111/2041-210X.12504

Judd, C. M., Westfall, J. and Kenny, D. A. (2017). Experiments with more than one random factor: Designs, analytic models, and statistical power. *Annual Review of Psychology*, 68, 601-625. doi:10.1146/annurevpsych122414033702

Kumle, L., Võ, M. L.-H. and Draschkow, D. (2021). Estimating power in (generalized) linear mixed models: An open introduction and tutorial in R. *Behavior Research Methods*, 53(6), 2528-2543. doi:10.3758/s13428021015460

Perugini, M., Gallucci, M. and Costantini, G. (2014). Safeguard power as a protection against imprecise power estimates. *Perspectives on Psychological Science*, 9(3), 319-332. doi:10.1177/1745691614528519

See Also

`simulate_design()` to simulate from the recovered specification, `power_mixed()` to run the power analysis it was read off the fit for and `model_formula()` for the analysis model a specification implies.

Examples

```
if (requireNamespace("lme4", quietly = TRUE)) {
  # Stand in for a pilot study: simulate a small design and fit it.
  pilot <- build_spec(list(name = "pilot", seed = 1, design_kind = "within",
    include_items = TRUE, n_subject = 30, n_item = 20, factor_name = "cond",
    lev1 = "a", lev2 = "b", intercept = 6, effect = 0.05,
    subj_int_sd = 0.12, subj_slope_sd = 0, subj_corr = 0,
    item_int_sd = 0.08, item_slope_sd = 0, item_corr = 0,
    family = "shifted_lognormal", resp_name = "", sigma = 0.3, shift = 200))
  fit <- lme4::lmer(model_formula(pilot), data = model_data(pilot, simulate_design(pilot)))

  # Read the design back off the fit, scaled up to the sample size being planned.
  spec <- spec_from_model(fit, n_subject = 60, n_item = 40)
  spec$random$subject$intercept_sd
  attr(spec, "column_kinds")
  head(simulate_design(spec))

  # The effect is the pilot's estimate. Plan for a smallest effect size of interest, here 0.03
  # on the log scale, or for the bound of the estimate's 60% interval nearer zero.
  spec_from_model(fit, n_subject = 60, n_item = 40, sesoi = c(effect = 0.03))$fixed
  spec_from_model(fit, n_subject = 60, n_item = 40, effects = "safeguard")$fixed
}
```

spec_json *Serialise a design specification to pretty-printed JSON*

Description

Serialise a design specification to pretty-printed JSON

Usage

```
spec_json(spec)
```

Arguments

spec	A design specification (list), as produced by build_spec().

Details

Each number is written at the shortest width that reads back as exactly the same IEEE-754 double, which is 15 significant digits for most values a user types and at most 17 for the rest. The JSON file is the portable artefact that the R and 'Python' implementations both read, so a fixed lower precision makes the specification itself a source of cross-language divergence: at 15 digits throughout, a coefficient of 1/3 came back as 0.33333333333333298, and over a sample of 214 doubles 189 failed to round-trip. Writing every number at 17 instead would round-trip but read badly, turning an effect typed as 0.3 into 0.29999999999999999.

A width is accepted only when 'jsonlite' reads it back as exactly the double it was written from. load_spec() reads the file with 'jsonlite', which parses numbers with the C library's strtod(), and the 'Python' package reads it with json. Both round correctly, so the two packages recover the same double. R's as.numeric() would not do as that check: on a build with long-double arithmetic, such as R on x86 Windows and Linux, it rounds a decimal twice, first to long double and then to double, and accepts some 15- and 16-digit forms that the readers of the file put one unit in the last place away.

Value

A length-one character string containing the specification as pretty-printed JSON, the portable artefact that the R and 'Python' packages both consume.

Examples

```
spec <- build_spec(list(name = "demo", seed = 1, design_kind = "between",
  factor_name = "group", lev1 = "a", lev2 = "b", n_subject = 20,
  intercept = 0, effect = 0.5, family = "gaussian", resp_name = "", sigma = 1))
cat(spec_json(spec))
```

sweep_spec *Sweep an analysis over one field of a design specification*

Description

Vary a single field of a specification across a set of values, run an analysis at each and bind the results into one data frame. Sample size is the axis users sweep most often, and `power_curve_mixed()` and `precision_curve()` call this function over it, but any field can be swept, including an effect size, a random-effect standard deviation, a residual standard deviation or the number of items per subject.

Usage

```
sweep_spec(
  spec,
  path,
  values,
  fn,
  ...,
  .name = NULL,
  .seeds = c("independent", "shared")
)
```

Arguments

`spec`	A design specification (path or list).
`path`	The field to vary, as `"units$subject$n"` or `c("units", "subject", "n")`.
`values`	A vector or list of values to set the field to, one grid point each. `design_conditions()` builds such a list for an effect-size sweep.
`fn`	The analysis to run at each grid point, for example `power_mixed()` or `precision_design()`. It is called as `fn(spec, ...)`.
`...`	Further arguments passed to `fn` at every grid point. A cluster passed as `cl` to an analysis that takes one, such as `power_mixed()`, serves every point, so that one pool of workers runs the whole sweep.
`.name`	Name for the column recording the swept value. Defaults to the last element of `path`.
`.seeds`	How the grid points are seeded. `"independent"`, the default, gives point `i` of `J` the seed `replicate_seeds(spec$seed, J)[i]`, and `"shared"` gives every point the specification's own seed (see Details).

Details

The specification is validated once, before the sweep, so a mistake in it is reported before any fitting starts rather than repeated at every grid point.

A value may be a scalar, which replaces the addressed field, or a list, which replaces it wholesale. The conditions `design_conditions()` builds are the exception, because each names only the effects it varies. Each condition is merged into the addressed list with `utils::modifyList()`, so the coefficients it names take its values and every other coefficient keeps the specification's, in the specification's order. That is the order in which the linear predictor is summed, so a condition simulates exactly the data of the specification edited by hand. A condition may name only coefficients the specification already has, and any other name is refused.

Each grid point draws its own replicates. Point `i` of a sweep over `J` values takes `replicate_seeds(spec$seed, J)[i]` as its seed, and the analysis derives its replicates' seeds from that, so no two points share their random numbers. `solve_curve()` fits the points of a curve as independent binomial counts, and its interval keeps its nominal coverage only when they are. The rule is prefix-stable: appending values to a sweep leaves the earlier points' results as they were. It depends on order, though, since a point's seed follows its position in `values`, so reordering the values, or inserting one, changes the results at every point that moves. A sweep over `seed` itself sets each point's seed to the value swept.

`.seeds = "shared"` gives every point the specification's own seed. The points then share their noise, so the curve looks smoother than its Monte Carlo error allows and an interval fitted to it is too narrow. Over 1,000 emulated curves for each of twelve settings on a tight grid of sample sizes, the nominal 95% interval of `solve_curve()` covered the true size in 87.8% of solves with shared seeds and in 96.5% with independent ones. On an effect-size axis, where the design and so every draw stay the same from point to point, the figures were 70.3% and 95.7%.

The result of `fn` is coerced to a data frame: a `pilotr_power` object becomes one row per focal effect, a data frame is used as it stands and a plain list becomes a single row. The swept value is added as a leading column named by `.name`. A list has no single value to record, so a sweep over lists, the conditions `design_conditions()` builds included, records the grid index there. `solve_curve()` then needs `x` to name the column that holds the effect. To vary a single effect, sweep its own coefficient. For example, `sweep_spec(spec, "fixed$coefficients$cond", c(0, 0.02, 0.04), power_mixed)` keeps the other coefficients and names the leading column `cond`, so `solve_curve(curve, target = 0.8, effect = "cond", transform = "identity")` returns the minimum detectable effect.

Value

A data frame binding the results, with the swept value as the leading column, or the grid index when the values are lists. `solve_curve()` reads that leading column by default, so a sweep over single values goes straight into a solve.

See Also

`design_conditions()` to build effect-size grids, `power_mixed()` and `precision_design()` for the analyses usually swept, and `solve_curve()` to solve the resulting curve for the swept value that meets a target.

Examples

```
if (requireNamespace("lme4", quietly = TRUE) &&
    requireNamespace("lmerTest", quietly = TRUE)) {
  spec <- build_spec(list(name = "p", seed = 1, design_kind = "within",
    include_items = TRUE, n_subject = 12, n_item = 12, factor_name = "cond",
    lev1 = "a", lev2 = "b", intercept = 6, effect = 0.05,
```

```
    subj_int_sd = 0.12, subj_slope_sd = 0.04, subj_corr = 0.2,
    item_int_sd = 0.08, item_slope_sd = 0.02, item_corr = -0.1,
    family = "shifted_lognormal", resp_name = "", sigma = 0.3, shift = 200))

  # Sample size, the same sweep power_curve_mixed() performs.
  sweep_spec(spec, "units$subject$n", c(12, 18), power_mixed, n_sims = 8)

  # Effect size, which power_curve_mixed() and precision_curve() do not sweep.
  sweep_spec(spec, "fixed$coefficients",
             design_conditions(effect = c(0.03, 0.06)), power_mixed, n_sims = 8)
}
```

`target_n` *Solve a power curve for the sample size that reaches a target power*

Description

The sample-size case of `solve_curve()`, and the number a power analysis is usually run to obtain. Takes the curve a sweep over sample size has produced and returns the size at which power reaches `target`, rounded up to a whole number of units alongside the exact solution.

Usage

```
target_n(curve, target = 0.8, ...)
```

Arguments

`curve`	A power curve, as returned by `power_curve_mixed()` or by `sweep_spec()` over `units$subject$n`.
`target`	The power to reach. Defaults to 0.8, the convention this package's plots draw a line at.
`...`	Further arguments passed to `solve_curve()`, such as `effect` to pick one focal effect out of a curve holding several, or `level` for the interval.

Details

Everything `solve_curve()` does applies here, including its refusals: a curve that never reaches the target within the sizes it swept is refused outright, and the reported interval can extend past the largest size simulated, which means the sweep was too narrow to settle the question.

The whole-number fields round up rather than to nearest, because a design cannot recruit a fraction of a subject and rounding down would leave the study short of the target it was sized for.

Value

The list `solve_curve()` returns, with `n`, `n_lo` and `n_hi` added: `value`, `lo` and `hi` rounded up to whole numbers.

See Also

solve_curve(), which this wraps, and power_curve_mixed() for the curve.

Examples

```
spec <- build_spec(list(name = "s", seed = 1, design_kind = "between", n_subject = 40,
  factor_name = "group", lev1 = "a", lev2 = "b", intercept = 0, effect = 0.7,
  family = "gaussian", resp_name = "score", sigma = 1))
# n_sims is small so the example runs quickly. Use 200 or more for real planning.
curve <- sweep_spec(spec, "units$subject$n", c(20, 40, 60, 80), power_design, n_sims = 50)
solved <- target_n(curve)
unlist(solved[c("n", "n_lo", "n_hi")])
```

---

validate_spec *Validate a design specification*

Description

Check a design specification against the portable schema and against the cross-field rules the schema cannot express, and check that its declared spec_version is one this implementation understands. Called by load_spec() by default.

Usage

```
validate_spec(spec, strict = TRUE)
```

Arguments

spec	A design specification (path or list).
strict	Whether an unrecognised field is an error (the default) or a warning. Set FALSE to load a specification carrying private annotations, accepting that a misspelled field will then be ignored in silence.

Details

Validation exists because several ways of getting a specification wrong produce plausible data and no error at all. A mistyped coefficient key resolves to no column and so silently sets that effect to zero, which generates exactly the data of a null design and reports success. A response parameter left over from another family is ignored. Neither is detectable in the output, which is why they are refused here.

Names are checked together as well as one by one. Two columns with one name, a contrast column defined by two factors or named like another column, an interaction whose analysis column already exists, a level listed twice, a correlation that pairs a term with itself or gives one pair twice, a factor both between and within a unit and a random.item entry in a design without items are each refused, since each would otherwise validate and change the data in silence. The rules are set out under Names in the specification. A within factor whose vary_within leaves out a unit of the design

draws a warning in either mode, since pilotr crosses it with every unit anyway, and a specification declaring version 0.4 is refused for it.

Several factors may be between one unit only in a specification declaring version 0.4 or later, which allocates them jointly, one block of units per combination of their levels. Up to version 0.3, pilotr assigned the levels of each between factor to blocks of its unit on its own, so two between factors over one unit fell into the same or overlapping blocks. A 2 x 2 between-subjects design over 40 subjects gave cells of 20, 0, 0 and 20, and its effects could not be estimated apart, so a specification declaring an earlier version is refused for them. Every version reads the same design written as one between factor whose levels are the cells, as the specification shows under Worked encodings, and `spec_from_model()` builds that factor from a fitted pilot.

From version 0.4, a within factor that sets `counterbalance` shows each subject each item at one level, rotating the levels over subjects and items as a Latin square does. It needs an item unit, and it is refused with `per_subject`, since every subject has to see every item. A warning says when the subjects or the items are not a multiple of the number of lists, which is the number of combinations of the counterbalanced factors' levels, so that the lists are unequal. The simulated data name each subject's list in a column `list`, so no other column of such a design may take that name.

Version negotiation covers the other direction. A specification that uses a feature introduced in 0.3 is read differently by a 0.2 implementation, so it must declare 0.3 or later, and one that uses a feature introduced in 0.4 must declare 0.4. A specification declaring a version newer than this implementation is refused outright. A specification with no `spec_version` is treated as 0.2, which is what every specification written before the field existed is.

Every field is read by its exact name, as the 'Python' package reads it, and a list that repeats a name within one object is refused, as `load_spec()` refuses a file that does. A seed must be a whole number within plus or minus 2^53 - 1, the range in which both packages read a JSON integer exactly.

Value

The specification, invisibly, so that the call can be chained.

Examples

```
spec <- build_spec(list(name = "demo", seed = 1, design_kind = "between",
  factor_name = "group", lev1 = "a", lev2 = "b", n_subject = 20,
  intercept = 0, effect = 0.5, family = "gaussian", resp_name = "", sigma = 1))
validate_spec(spec)

# A mistyped coefficient key is refused rather than passed as a zero effect.
bad <- spec
bad$fixed$coefficients <- list(effct = 0.5)
try(validate_spec(bad))
```

Index

as241, 4
as241(), *3*

brms_bridge, 5
brms_bridge(), *3*, *7*, *12–14*, *27*, *38*
build_spec, 7
build_spec(), *3*, *15*, *44*, *54*

calibrate_response, 8
calibrate_response(), *42*

default_response_name, 10
default_response_name(), *3*, *7*
design_conditions, 11
design_conditions(), *21, 22*, *25*, *28*, *32*, *34*, *46*, *50*, *55, 56*

generate_design_analysis, 12
generate_design_analysis(), *15, 16*, *27*, *37*
generate_r_script, 15
generate_r_script(), *3*, *14*, *16*

load_spec, 16
load_spec(), *3*, *15*, *18–20*, *44*, *54*, *58, 59*

make_rng, 17
make_rng(), *3*
model_data, 18
model_data(), *3*, *5*, *18, 19*, *26, 27*, *33*, *36*, *51, 52*
model_formula, 18
model_formula(), *3*, *6*, *12*, *26, 27*, *33*, *35*, *53*

parallel::makeCluster(), *24*, *26*, *36*
pilotr *(*pilotr-package*)*, 3
pilotr-package, 3
pilotr_example, 20
pilotr_example(), *3*
power_curve_mixed, 20
power_curve_mixed(), *3*, *45–47*, *55*, *57, 58*
power_design, 23
power_design(), *3*, *15*, *30, 31*, *43*
power_mixed, 25
power_mixed(), *3*, *15*, *19*, *21, 22*, *24*, *30–32*, *35–37*, *39*, *43*, *46*, *50*, *52, 53*, *55, 56*
power_report, 30
precision_curve, 33
precision_curve(), *3*, *45–47*, *55*
precision_design, 35
precision_design(), *3*, *14*, *19*, *30–34*, *52*, *55, 56*
print.pilotr_bridge, 38
print.pilotr_bridge(), *6*
print.pilotr_power, 39
print.pilotr_report *(*power_report*)*, 30

replicate_seeds, 39
replicate_seeds(), *23*, *36*
response_variance, 41
response_variance(), *5*, *9, 10*
run_app, 43
run_app(), *3*

shiny::runApp(), *43*
simulate_design, 44
simulate_design(), *3*, *5, 6*, *8*, *16*, *18*, *20*, *50*, *52, 53*
solve_curve, 45
solve_curve(), *31*, *33, 34*, *56–58*
spec_from_model, 48
spec_from_model(), *59*
spec_json, 54
spec_json(), *3*, *7, 8*, *15*, *52*
stats::contrasts(), *50*
stats::formula, *19*
stats::power.t.test(), *46*
stats::qnorm(), *4*
stats::terms(), *5*
sweep_spec, 55

sweep_spec(), *11*, *20–22*, *24*, *26*, *30*, *34*, *36*, *45–47*, *50*, *57*

target_n, 57
target_n(), *3*, *20–22*, *30–32*, *47*

utils::modifyList(), *56*

validate_spec, 58
validate_spec(), *16*, *44*, *51*, *52*