Package 'pilotr'

September 20, 2026

Title Simulate Experimental and Behavioural Data from a Portable Design Specification

Version 0.3.1

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: exactly for the Gaussian family and for any family with rounding set, and to within the last unit in the last place for families applying a transcendental function to the linear predictor, whose rounding the IEEE-754 standard does not fix.

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

Depends R (>= 4.0.0)

Imports jsonlite, parallel, stats

Suggests shiny, future, promises, ggplot2 (>= 3.4.0), lme4, lmerTest, callr, knitr, rmarkdown, testthat (>= 3.0.0), MASS

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 2026-09-20 21:50:02 UTC

Contents

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. Fixed effect sizes are user-specified, by-subject and by-item random intercepts and slopes are crossed, and the response families cover Gaussian, lognormal, shifted lognormal, 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 one ready-to-run specification per design family, 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 twin 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.

For analysis, `model_data()` adds the response column that the model expects, and `model_formula()` derives the maximal mixed-model formula the design implies. `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, 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 to locate where a design becomes adequately powered. `precision_design()` and its curve counterpart `precision_curve()` report the width of the interval a design buys and the decision probabilities against a region of practical equivalence.

Two functions round the package off. `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).

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).

Value

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

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 spec (path or list).
prior_scale	SD of the Normal prior on fixed main effects (standardised scale).
interaction_scale	SD of the Normal prior on interaction terms (default prior_scale/2).

Value

An object of class `pilotr_bridge`: a list with elements `formula`, `family`, `priors`, and `code`, the last being a ready-to-fit `brms` model. 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.

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`, `n_item`, `factor_name`, `lev1`, `lev2`, `intercept`, `effect`, `family`, `resp_name`, and family parameters such as `sigma`, `shift`, `thresholds`, or `phi`; within-design random effects use `subj_int_sd`, `subj_slope_sd`, `subj_corr`, `item_int_sd`, `item_slope_sd`, and `item_corr`.

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.

Usage

```
calibrate_response(spec, target_var = 1, tune = c("sigma", "all"))
```

Arguments

`spec`	A design specification (path or list).
`target_var`	The total variance to calibrate to. 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.

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"` multiplies the intercept, 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 calibrating one to a total variance of 1 is not merely difficult but impossible, 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 every term by `k` multiplies each row's linear predictor by exactly `k`.

Value

The specification, with the tuned parameters replaced.

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)
```

---

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"`, `"bernoulli"`, `"poisson"`, `"ordinal"`, or `"beta"`.

Value

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

Examples

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

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

Description

Produce the list of `fixed$coefficients` objects 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 hold fixed in every condition, for effects the sweep does not vary.

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.

Any coefficient the specification has that is not named here is left at its own value, so a sweep varies only the effects it names.

Value

A list of named lists, each suitable as a `fixed$coefficients` value.

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 coefficients of the emitted model, so an interaction is written `a:b` as in the specification, not `a_b` as in the `lme4` formula from `model_formula()`. 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 is written for a generic SLURM cluster and carries two placeholders, marked `EDIT` in its header, that must be filled in before submission: the `#SBATCH --account` directive and the writable `PROJECT_DIR`. 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 a small pilot most often deserves.

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 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 conclusion from a fit that has not converged carries the authority of a number without the sampling behind it.

A verdict is a function of the whole rule, the whole gate and the specification, so each replicate's record 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 is not submittable as emitted, since its `--account` and `PROJECT_DIR` placeholders must be filled in first.

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 Burkner, 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]], 15), 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 an R list literal, 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 confirms that it reproduces the data bit-for-bit.

Usage

```
generate_r_script(spec)
```

Arguments

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

Details

Numbers are emitted at the shortest width that reads back as the same double, rather than through `deparse()`, which prints 15 significant digits and so does not round-trip: `deparse(1/3)` reads back as a different double. Since the point of the script is bit-for-bit reproduction, the embedded specification has to preserve every coefficient exactly.

Value

A length-one character string containing a runnable R script that loads `pilotr`, embeds the specification, and simulates the data.

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.
validate	Whether to validate the specification after reading it. TRUE (the default) applies validate_spec() with strict = TRUE; FALSE skips validation, and any other value is passed to validate_spec() as its strict argument.

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.

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. The draw-order contract is 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.

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-transformed for the lognormal families), 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).
`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)
```

Arguments

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

Value

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

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)
```

pilotr_example *Example design specifications shipped with pilotr*

Description

pilotr ships one ready-to-run specification per design family, as JSON, in the package's `examples/` directory. These are the same files that drive the Python twin and the no-code app, so a design authored in one place 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. A thin wrapper around sweep_spec() over units$subject$n, kept because a sample-size curve is the sweep users want most often. Requires the lme4 and lmerTest packages.

Usage

```
power_curve_mixed(
  spec,
  subject_ns,
  focal = NULL,
  n_sims = 60,
  alpha = 0.05,
  workers = 1
)
```

Arguments

spec	A design specification (path or list).
subject_ns	A numeric vector of subject counts to evaluate.
focal	The fixed effects to test, 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` Two-sided significance level.

`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.

Details

Like `power_mixed()`, this runs pilotr's own simulation loop over the portable design specification rather than wrapping an existing package, and differs from `simr` (Green and MacLeod, 2016) and `mixedpower` (Kumle, Vo and Draschkow, 2021) in being driven by that specification, in reporting Type S and Type M errors, and in built-in parallelisation: with `workers > 1` a single worker pool is created once and reused across all sample sizes.

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.

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_m`, and the `n_attempted`, `n_returned`, `n_converged`, `n_singular` and `n_warning` fit counts. `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

Kumle, L., Vo, 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, 2528-2543. doi:10.3758/s13428021015460

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)
```

Arguments

`spec`	A design specification (path or list) for a two-group Gaussian design.
`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`	Two-sided significance level.
`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.

Value

A list with elements `n_sims`, `alpha`, `power`, `n_significant`, `true_effect`, `mean_estimate`, `type_s` (sign-error rate among significant replicates), and `type_m` (mean exaggeration ratio among significant replicates). 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.

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

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 with `lmerTest`, and test each focal fixed effect using Satterthwaite p-values. 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
)
```

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`.
`formula`	Optional `lme4` formula; if `NULL` it is derived from the specification via `model_formula()`.

prep	Optional function mapping a simulated data set to the modelling data; if NULL it is derived via 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	Two-sided significance level.
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.

Details

power_mixed() is not a wrapper around an existing power package: it runs pilotr's own simulation loop over the portable design specification. It covers territory pioneered by simr (Green and MacLeod, 2016) and mixedpower (Kumle, Vo and Draschkow, 2021), to which it is indebted. pilotr differs in being driven by the portable cross-language specification, in reporting the Type S and Type M design-analysis errors alongside power, and in parallelising its replicates through the workers argument.

The analysis model comes from the specification rather than from this function. Before 0.3 the formula was written into the source as a maximal crossed structure, so a design declaring uncorrelated slopes, or no slopes at all, was nonetheless analysed as though it had them, and a design with more than one factor was refused. The formula now comes from model_formula() and the data from model_data(), so the analysis matches the process that generated the data. Both can still be given directly, which is what to do when a deliberately different analysis model is the point, as when checking how a misspecified model behaves.

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.

Value

An object of class pilotr_power, a list whose per-run elements are n_sims, alpha, 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 and type_m. With a single focal effect each of those has length one, so result$power reads as it always has.

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. 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.

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

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., Vo, 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, 2528-2543. doi:10.3758/s13428021015460

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

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)
}
```

`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
)
```

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 via `model_formula()`.
`prep`	Optional data-preparation function; if `NULL` it is derived via `model_data()`.
`rope`	Half-width of the region of practical equivalence. Set it clearly narrower than the smallest effect worth detecting, because the probability of a determinate meaningful decision about an effect no larger than `rope` 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.

Details

A thin wrapper around `sweep_spec()` over `units$subject$n`, kept because a sample-size curve is the sweep users want most often. 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.

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, adding an `n_subject` column to 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.

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 to zero), 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
)
```

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.
formula	Optional lme4 formula; if NULL it is derived from the specification via model_formula().
prep	Optional function mapping a simulated data set to the modelling data; if NULL it is derived via model_data(), which log-transforms the outcome and builds the contrast and interaction columns, so focal names follow the auto-formula (interactions written as a_b).
rope	Half-width of the region of practical equivalence; an effect with abs(beta) < rope is treated as practically equivalent to zero. Set it clearly narrower than the smallest effect worth detecting, because the probability of a determinate meaningful decision about an effect no larger than rope 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 converged replicates, 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.

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.

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, and n_warning. 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.

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. 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).

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

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 and then one row per focal effect, with 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.

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 significant digits for the reported rates.
...	Ignored, present for consistency with the generic.

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

Until 0.3 the rule was `base + (i - 1)`. Consecutive seeds are not independent streams in this generator: seeding sets `s1` to `1 + (seed mod 2147483562)` and `s2` from `s1`, and only ten warm-up draws are discarded, so replicate `i` and replicate `i + 1` begin a few steps apart in the same sequence rather than in unrelated parts of it. Measured over 2,000 replicates, the first draw of replicate `i` correlated 0.95 with the first draw of replicate `i + 1`.

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 itself linear in the seed, the first draws remained correlated at -0.27. The problem is linearity, so no linear remedy addresses it.

Drawing the seeds from the shared generator does work. Successive outputs of the combined generator are what that generator exists to make look independent, so the seeds inherit it: 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, and the skipping is deterministic, so the R and 'Python' implementations still agree.

This changed every number pilotr produced before 0.3, and the first replicate no longer uses the specification's own seed. Both are deliberate; pin an earlier version to reproduce earlier output.

Value

A numeric vector of `n` distinct seeds.

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, which matters because drawing from it and estimating a variance from the drawn effects of, say, 30 subjects carries a sampling error of around 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 the linear predictor lives on, which is what makes them comparable and what makes their total the right denominator for an effect size. 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.

Value

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

References

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

See Also

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

Examples

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

# 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 work is not blocked by a shared process, and power simulations can be parallelised across the user's own cores.

Usage

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

Arguments

...	passed to shiny::runApp() (e.g. port, launch.browser).
async	If TRUE (default when 'future' and 'promises' are installed), set a multisession future plan so power runs execute in a background worker and do not block the UI. This keeps the app responsive on your own machine while a long power run completes in the background.

Value

No return value; 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.

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.

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 point read off a simulated curve without one repeats the overconfidence that design analysis exists to expose.

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.
`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 2.9% of the analytic answer on average, against 3.4% for a logit fitted the same way.

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 the honest shape: at the top of a power curve a given change in rate costs far more sample size than the same change lower down. 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.

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.

What the interval covers is the Monte Carlo uncertainty of the fit, not the gap between the fitted shape and the true curve. Across 36 checks against stats::power.t.test() at 400 replicates a point, the solved size sat within 2.9% of the analytic answer on average and within 6.9% at worst, and the nominal 95% interval covered the analytic value 35 times out of 36. 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 a finer grid, more replicates, or a design with a closed form to check against.

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 deserves a wider grid or more replicates before it deserves any trust.

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.

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, with no numbers left to invent. Requires the `lme4` package.

Usage

```
spec_from_model(
  fit,
  name = NULL,
  seed = 1,
  n_subject = NULL,
  n_item = NULL,
  family = NULL,
  round = NULL
)
```

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 actually had, and is normally raised above it, since scaling a pilot design up is the point of reading a specification off a pilot fit.
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.

Details

Settling on 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, and the tutorials on the method converge on the same remedy, 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 performs that transfer, and the specification it returns can be enlarged and passed straight to `simulate_design()` or `power_mixed()`.

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 lives 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 nothing more, it is reported by a message on every call, and 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.

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 boundary-singular pilot fit is carried across as it stands, which means a variance estimated at zero or a correlation estimated at exactly plus or minus one. Those are faithful readings of the fit, and no kind of defect, but they are also the sign that the random-effect structure was richer than the pilot could support, and a power analysis resting on them will inherit that (Bates et al., 2015). Widening the design, or simplifying the structure before refitting, is the remedy.

Value

A design specification as a nested list, carrying `spec_version` 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"`. Both are attributes rather than fields so that the specification itself stays within the portable schema.

References

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

Kumle, L., Vo, 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, 2528-2543. doi:10.3758/s13428021015460

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))
}
```

`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 0.3333333333333298, 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.

Preferring the shorter width is not only a matter of appearance. A build of R without long-double arithmetic reads 17 significant digits inexactly, because the digits overflow the 53-bit mantissa before the decimal exponent is applied, so the few values that need that width are the ones such a build cannot recover. Anything shorter it reads correctly.

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() are wrappers around this for 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)
```

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.
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.
.name	Name for the column recording the swept value. Defaults to the last element of path.

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. Replacing a whole fixed$coefficients object is how an effect-size sweep works, and design_conditions() builds those objects, including a common all-zero condition for examining behaviour under the null.

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 where the value is a scalar.

Value

A data frame binding the results, with the swept value as the leading column. When the swept values are not scalars, that column holds the grid index instead. solve_curve() reads that leading column, so a sweep 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 the old curve functions could not reach.
  sweep_spec(spec, "fixed$coefficients",
             design_conditions(effect = c(0, 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.

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. 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.

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, where it used to pass as a zero effect.
bad <- spec
bad$fixed$coefficients <- list(effct = 0.5)
try(validate_spec(bad))
```

Index