Package 'skpr'

April 25, 2025

Title Design of Experiments Suite: Generate and Evaluate Optimal Designs

Date 2025-04-07

Version 1.8.2

Description Generates and evaluates D, I, A, Alias, E, T, and G optimal designs. Supports generation and evaluation of blocked and split/split-split/.../N-split plot designs. Includes parametric and Monte Carlo power evaluation functions, and supports calculating power for censored responses. Provides a framework to evaluate power using functions provided in other packages or written by the user. Includes a Shiny graphical user interface that displays the underlying code used to create and evaluate the design to improve ease-of-use and make analyses more reproducible. For details, see Morgan-Wall et al. (2021) <doi:10.18637/jss.v099.i01>.

Copyright Institute for Defense Analyses

Depends R (>= 4.1.0)

License GPL-3

RoxygenNote 7.3.2

Imports utils, iterators, stats, lme4, Rcpp (>= 0.11.0), foreach, doParallel, survival, future, car, viridis, magrittr, lmerTest, methods, progress, doRNG, doFuture, progressr, geometry, digest

LinkingTo Rcpp, RcppEigen

Suggests testthat, mbest, ggplot2, lmtest, cli, gridExtra, rintrojs, shinythemes, shiny, shinyjs, gt, shinytest2

Encoding UTF-8

URL https://github.com/tylermorganwall/skpr, https://tylermorganwall.github.io/skpr/

BugReports https://github.com/tylermorganwall/skpr/issues

NeedsCompilation yes

Author Tyler Morgan-Wall [aut, cre], George Khoury [aut]

Maintainer Tyler Morgan-Wall <tylermw@gmail.com>

Repository CRAN

Date/Publication 2025-04-25 18:40:02 UTC

Contents

calculate_power_curves 2
contr.simplex 4
eval_design 5
eval_design_custom_mc 9
eval_design_mc 13
eval_design_survival_mc 19
gen_design 22
get_attribute 30
get_optimality 31
get_power_curve_output 32
plot_correlations 33
plot_fds 34
print.skpr_eval_output 35
print.skpr_power_curve_output 36
skprGUI 37
%>% 37

Index **38**

`calculate_power_curves` *Calculate Power Curves*

Description

Calculate and optionally plot power curves for different effect sizes and trial counts. This function takes a

Usage

```
calculate_power_curves(
  trials,
  effectsize = 1,
  candidateset = NULL,
  model = NULL,
  alpha = 0.05,
  gen_args = list(),
  eval_function = "eval_design",
  eval_args = list(),
  random_seed = 123,
  iterate_seed = FALSE,
  plot_results = TRUE,
  auto_scale = TRUE,
  x_breaks = NULL,
  y_breaks = seq(0, 1, by = 0.1),
  ggplot_elements = list()
)
```

Arguments

trials	A numeric vector indicating the trial(s) used when computing the power curve. If a single value, this will be fixed and only ‘effectsize‘ will be varied.
effectsize	Default ‘1‘. A numeric vector indicating the effect size(s) used when computing the power curve. If a single value, this will be fixed and only ‘trials‘ will be varied. If using a length-2 effect size with ‘eval_design_mc()‘ (such as a binomial probability interval), the effect size pairs can be input as entries in a list.
candidateset	Default ‘NULL‘. The candidate set (see ‘gen_design()‘ documentation for more information). Provided to aid code completion: can also be provided in ‘gen_args‘.
model	Default ‘NULL‘. The model (see ‘gen_design()‘ and ‘eval_design()‘ documentation for more information). Provided to aid code completion: can also be provided in ‘gen_args‘/‘eval_args‘.
alpha	Default ‘0.05‘. The allowable Type-I error rate (see ‘eval_design()‘ documentation for more information). Provided to aid code completion: can also be provided in ‘eval_args‘.
gen_args	Default ‘list()‘. A list of argument/value pairs to specify the design generation parameters for ‘gen_design()‘.
eval_function	Default ‘"eval_design"‘. A string (or function) specifying the skpr power evaluation function. Can also be ‘"eval_design_mc"‘, ‘"eval_design_survival_mc"‘, and ‘"eval_design_custom_mc"‘.
eval_args	Default ‘list()‘. A list of argument/value pairs to specify the design power evaluation parameters for ‘eval_function‘.
random_seed	Default ‘123‘. The random seed used to generate and then evaluate the design. The seed is set right before design generation.
iterate_seed	Default ‘FALSE‘. This will iterate the random seed with each new design. Set this to ‘TRUE‘ to add more variability to the design generation process.
plot_results	Default ‘TRUE‘. Whether to print out a plot of the power curves in addition to the data frame of results. Requires ‘ggplot2‘.
auto_scale	Default ‘TRUE‘. Whether to automatically scale the y-axis to 0 and 1.
x_breaks	Default ‘NULL‘, automaticly generated by ggplot2.
y_breaks	Default ‘seq(0,1,by=0.1)‘. Y-axis breaks.
ggplot_elements	Default ‘list()‘. Extra ‘ggplot2‘ elements to customize the plot, passed in as elements in a list.

Value

A data.frame of power values with design generation information.

Examples

```
if(skpr:::run_documentation()) {
cand_set = expand.grid(brew_temp = c(80, 85, 90),
                       altitude = c(0, 2000, 4000),
```

```
                      bean_sun = c("low", "partial", "high"))
#Plot power for a linear model with all interactions
calculate_power_curves(trials=seq(10,60,by=5),
                      candidateset = cand_set,
                      model = ~.*.,
                      alpha = 0.05,
                      effectsize = 1,
                      eval_function = "eval_design") |>
 head(30)

}
if(skpr:::run_documentation()) {
#Add multiple effect sizes
calculate_power_curves(trials=seq(10,60,by=1),
                      candidateset = cand_set,
                      model = ~.*.,
                      alpha = 0.05,
                      effectsize = c(1,2),
                      eval_function = "eval_design") |>
 head(30)
}
if(skpr:::run_documentation()) {
#Generate power curve for a binomial model
calculate_power_curves(trials=seq(50,150,by=10),
                      candidateset = cand_set,
                      model = ~.,
                      effectsize = c(0.6,0.9),
                      eval_function = "eval_design_mc",
                      eval_args = list(nsim = 100, glmfamily = "binomial")) |>
 head(30)
}
if(skpr:::run_documentation()) {
#Generate power curve for a binomial model and multiple effect sizes
calculate_power_curves(trials=seq(50,150,by=10),
                      candidateset = cand_set,
                      model = ~.,
                      effectsize = list(c(0.5,0.9),c(0.6,0.9)),
                      eval_function = "eval_design_mc",
                      eval_args = list(nsim = 100, glmfamily = "binomial")) |>
 head(30)
}
```

`contr.simplex` *Orthonormal Contrast Generator*

Description

Generates orthonormal (orthogonal and normalized) contrasts. Each row is the vertex of an N-dimensional simplex. The only exception are contrasts for the 2-level case, which return 1 and -1.

Usage

```
contr.simplex(n, size = NULL)
```

Arguments

n	The number of levels in the catagorical variable. If this is a factor or character vector, ‘n‘ will be ‘length(n)‘
size	Default ‘1‘. The length of the simplex vector.

Value

A matrix of Orthonormal contrasts.

Examples

```
contr.simplex(4)
```

---

eval_design *Calculate Power of an Experimental Design*

Description

Evaluates the power of an experimental design, for normal response variables, given the design's run matrix and the statistical model to be fit to the data. Returns a data frame of parameter and effect powers. Designs can consist of both continuous and categorical factors. By default, `eval_design` assumes a signal-to-noise ratio of 2, but this can be changed with the `effectsize` or `anticoef` parameters.

Usage

```
eval_design(
  design,
  model = NULL,
  alpha = 0.05,
  blocking = NULL,
  anticoef = NULL,
  effectsize = 2,
  varianceratios = NULL,
  contrasts = contr.sum,
  conservative = FALSE,
  reorder_factors = FALSE,
  detailedoutput = FALSE,
  advancedoptions = NULL,
  high_resolution_candidate_set = NA,
  moment_sample_density = 20,
  ...
)
```

Arguments

design	The experimental design. Internally, eval_design rescales each numeric column to the range [-1, 1], so you do not need to do this scaling manually.
model	The model used in evaluating the design. If this is missing and the design was generated with skpr, the generating model will be used. It can be a subset of the model used to generate the design, or include higher order effects not in the original design generation. It cannot include factors that are not present in the experimental design.
alpha	Default '0.05'. The specified type-I error.
blocking	Default 'NULL'. If 'TRUE', eval_design will look at the rownames (or blocking columns) to determine blocking structure. Default FALSE.
anticoef	The anticipated coefficients for calculating the power. If missing, coefficients will be automatically generated based on the effectsize argument.
effectsize	Default '2'. The signal-to-noise ratio. For continuous factors, this specifies the difference in response between the highest and lowest levels of the factor (which are -1 and +1 after eval_design normalizes the input data), assuming that the root mean square error is 1. If you do not specify anticoef, the anticipated coefficients will be half of effectsize. If you do specify anticoef, effectsize will be ignored.
varianceratios	Default 'NULL'. The ratio of the whole plot variance to the run-to-run variance. If not specified during design generation, this will default to 1. For designs with more than one subplot this ratio can be a vector specifying the variance ratio for each subplot (comparing to the run-to-run variance). Otherwise, it will use a single value for all strata.
contrasts	Default contr.sum. The function to use to encode the categorical factors in the model matrix. If the user has specified their own contrasts for a categorical factor using the contrasts function, those will be used. Otherwise, skpr will use 'contr.sum'.
conservative	Specifies whether default method for generating anticipated coefficents should be conservative or not. 'TRUE' will give the most conservative estimate of power by setting all but one (or multiple if they are equally low) level in each categorical factor's anticipated coefficients to zero. Default 'FALSE'.
reorder_factors	Default 'FALSE'. If 'TRUE', the levels will be reordered to generate the most conservative calculation of effect power. The function searches through all possible reference levels for a given factor and chooses the one that results in the lowest effect power. The reordering will be presenting in the output when 'detailedoutput = TRUE'.
detailedoutput	If 'TRUE", return additional information about evaluation in results. Default FALSE.
advancedoptions	Default 'NULL'. A named list with parameters to specify additional attributes to calculate. Options: 'aliaspower' gives the degree at which the Alias matrix should be calculated.

`high_resolution_candidate_set`
: Default 'NA'. If you have continuous numeric terms and disallowed combinations, the closed-form I-optimality value cannot be calculated and must be approximated by numeric integration. This requires sampling the allowed space densely, but most candidate sets will provide a sparse sampling of allowable points. To work around this, skpr will generate a convex hull of the numeric terms for each unique combination of categorical factors to generate a dense sampling of the space and cache that value internally, but this is a slow calculation and does not support non-convex candidate sets. To speed up moment matrix calculation, pass a higher resolution version of your candidate set here with the disallowed combinations already applied. If you generated your design externally from skpr, there are disallowed combinations in your design, and need correct I-optimalituy values, you must pass your candidate set here.

`moment_sample_density`
: Default '20'. The density of points to sample when calculating the moment matrix to compute I-optimality. Only required if the design was generated outside of skpr and there are disallowed combinations. It is much faster and potentially more accurate to provide your own candidate set with higher resolution continuous factors to 'high_resolution_candidate_set'.'

`...`
: Additional arguments.

Details

This function evaluates the power of experimental designs.

If the design is has no blocking or restrictions on randomization, the model assumed is:

$y = X\beta + \epsilon$.

If the design is a split-plot design, the model is as follows:

$y = X\beta + Zb_i + \epsilon_{ij}$,

Here, y is the vector of experimental responses, X is the model matrix, β is the vector of model coefficients, Z_i are the blocking indicator, b_i is the random variable associated with the ith block, and ϵ is a random variable normally distributed with zero mean and unit variance (root-mean-square error is 1.0).

`eval_design` calculates both parameter power as well as effect power, defined as follows:

1) Parameter power is the probability of rejecting the hypothesis $H_0 : \beta_i = 0$, where β_i is a single parameter in the model 2) Effect power is the probability of rejecting the hypothesis $H_0 : \beta_1 = \beta_2 = ... = \beta_n 0$ for all n coefficients for a categorical factor.

The two power types are equivalent for continuous factors and two-level categorical factors, but they will differ for categorical factors with three or more levels.

For split-plot designs, the degrees of freedom are allocated to each term according to the algorithm given in "Mixed-Effects Models in S and S-PLUS" (Pinheiro and Bates, pp. 91).

When using `conservative = TRUE`, `eval_design` first evaluates the power with the default (or given) coefficients. Then, for each multi-level categorical, it sets all coefficients to zero except the level that produced the lowest power, and then re-evaluates the power with this modified set of anticipated coefficients. If there are two or more equal power values in a multi-level categorical, two of the lowest equal terms are given opposite sign anticipated coefficients and the rest (for that categorical factor) are set to zero.

Value

A data frame with the parameters of the model, the type of power analysis, and the power. Several design diagnostics are stored as attributes of the data frame. In particular, the `modelmatrix` attribute contains the model matrix that was used for power evaluation. This is especially useful if you want to specify the anticipated coefficients to use for power evaluation. The model matrix provides the order of the model coefficients, as well as the encoding used for categorical factors.

Examples

```
#Generating a simple 2x3 factorial to feed into our optimal design generation
#of an 11-run design.
factorial = expand.grid(A = c(1, -1), B = c(1, -1), C = c(1, -1))

optdesign = gen_design(candidateset = factorial,
                       model= ~A + B + C, trials = 11, optimality = "D", repeats = 100)

#Now evaluating that design (with default anticipated coefficients and a effectsize of 2):
eval_design(design = optdesign, model= ~A + B + C, alpha = 0.2)

#Evaluating a subset of the design (which changes the power due to a different number of
#degrees of freedom)
eval_design(design = optdesign, model= ~A + C, alpha = 0.2)

#We do not have to input the model if it's the same as the model used
#During design generation. Here, we also use the default value for alpha (`0.05`)
eval_design(optdesign)

#Halving the signal-to-noise ratio by setting a different effectsize (default is 2):
eval_design(design = optdesign, model= ~A + B + C, alpha = 0.2, effectsize = 1)

#With 3+ level categorical factors, the choice of anticipated coefficients directly changes the
#final power calculation. For the most conservative power calculation, that involves
#setting all anticipated coefficients in a factor to zero except for one. We can specify this
#option with the "conservative" argument.

factorialcoffee = expand.grid(cost = c(1, 2),
                         type = as.factor(c("Kona", "Colombian", "Ethiopian", "Sumatra")),
                               size = as.factor(c("Short", "Grande", "Venti")))

designcoffee = gen_design(factorialcoffee,
                       ~cost + size + type, trials = 29, optimality = "D", repeats = 100)

#Evaluate the design, with default anticipated coefficients (conservative is FALSE by default).
eval_design(designcoffee)

#Evaluate the design, with conservative anticipated coefficients:
eval_design(designcoffee, conservative = TRUE)

#which is the same as the following, but now explicitly entering the coefficients:
eval_design(designcoffee, anticoef = c(1, 1, 1, 0, 0, 1, 0))

#You can also evaluate the design with higher order effects, even if they were not
```

```
#used in design generation:
eval_design(designcoffee, model = ~cost + size + type + cost * type)


#Generating and evaluating a split plot design:
splitfactorialcoffee = expand.grid(caffeine = c(1, -1),
                                   cost = c(1, 2),
                           type = as.factor(c("Kona", "Colombian", "Ethiopian", "Sumatra")),
                                   size = as.factor(c("Short", "Grande", "Venti")))

coffeeblockdesign = gen_design(splitfactorialcoffee, ~caffeine, trials = 12)
coffeefinaldesign = gen_design(splitfactorialcoffee,
                               model = ~caffeine + cost + size + type, trials = 36,
                               splitplotdesign = coffeeblockdesign, blocksizes = 3)

#Evaluating design (blocking is automatically detected)
eval_design(coffeefinaldesign, 0.2, blocking = TRUE)

#Manually turn blocking off to see completely randomized design power
eval_design(coffeefinaldesign, 0.2, blocking = FALSE)

#We can also evaluate the design with a custom ratio between the whole plot error to
#the run-to-run error.
eval_design(coffeefinaldesign, 0.2, varianceratios = 2)

#If the design was generated outside of skpr and thus the row names do not have the
#blocking structure encoded already, the user can add these manually. For a 12-run
#design with 4 blocks, here is a vector indicated the blocks:

blockcolumn = c(1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4)

#If we wanted to add this blocking structure to the design coffeeblockdesign, we would
#add a column with the format "Block1", "Block2", "Block3" ... and each one will be treated
#as a separate blocking layer.

coffeeblockdesign$Block1 = blockcolumn

#By default, skpr will throw out the blocking columns unless the user specifies `blocking = TRUE`.
eval_design(coffeeblockdesign, blocking=TRUE)
```

`eval_design_custom_mc` *Monte Carlo power evaluation for experimental designs with user-supplied libraries*

Description

Evaluates the power of an experimental design, given its run matrix and the statistical model to be fit to the data, using monte carlo simulation. Simulated data is fit using a user-supplied fitting library and power is estimated by the fraction of times a parameter is significant. Returns a data frame of parameter powers.

Usage

```
eval_design_custom_mc(
  design,
  model = NULL,
  alpha = 0.05,
  nsim,
  rfunction,
  fitfunction,
  pvalfunction,
  anticoef,
  effectsize = 2,
  contrasts = contr.sum,
  coef_function = coef,
  calceffect = FALSE,
  detailedoutput = FALSE,
  parameternames = NULL,
  advancedoptions = NULL,
  progress = TRUE,
  parallel = FALSE,
  parallel_pkgs = NULL,
  ...
)
```

Arguments

design	The experimental design. Internally, eval_design_custom_mc rescales each numeric column to the range [-1, 1].
model	The model used in evaluating the design. If this is missing and the design was generated with skpr, the generating model will be used. It can be a subset of the model used to generate the design, or include higher order effects not in the original design generation. It cannot include factors that are not present in the experimental design.
alpha	Default '0.05'. The type-I error. p-values less than this will be counted as significant.
nsim	The number of simulations.
rfunction	Random number generator function. Should be a function of the form f(X, b), where X is the model matrix and b are the anticipated coefficients.
fitfunction	Function used to fit the data. Should be of the form f(formula, X, contrasts) where X is the model matrix. If contrasts do not need to be specified for the user supplied library, that argument can be ignored.
pvalfunction	Function that returns a vector of p-values from the object returned from the fitfunction.
anticoef	The anticipated coefficients for calculating the power. If missing, coefficients will be automatically generated based on effectsize.
effectsize	The signal-to-noise ratio. Default 2. For a gaussian model, and for continuous factors, this specifies the difference in response between the highest and

lowest levels of a factor (which are +1 and -1 after normalization). More precisely: If you do not specify `anticoef`, the anticipated coefficients will be half of `effectsize`. If you do specify `anticoef`, `effectsize` will be ignored.

`contrasts` Default `contr.sum`. Function used to generate the contrasts encoding for categorical variables. If the user has specified their own contrasts for a categorical factor using the contrasts function, those will be used. Otherwise, skpr will use contr.sum.

`coef_function` Function that, when applied to a fitfunction return object, returns the estimated coefficients.

`calceffect` Default 'FALSE'. Calculates effect power for a Type-III Anova (using the car package) using a Wald test. this ratio can be a vector specifying the variance ratio for each subplot. Otherwise, it will use a single value for all strata. To work, the fit returned by 'fitfunction' must have a method compatable with the car package.

`detailedoutput` Default 'FALSE'. If 'TRUE', return additional information about evaluation in results.

`parameternames` Vector of parameter names if the coefficients do not correspond simply to the columns in the model matrix (e.g. coefficients from an MLE fit).

`advancedoptions`

Default 'NULL'. Named list of advanced options. 'advancedoptions$anovatype' specifies the Anova type in the car package (default type 'III'), user can change to type 'II'). 'advancedoptions$anovatest' specifies the test statistic if the user does not want a 'Wald' test–other options are likelyhood-ratio 'LR' and F-test 'F'. 'advancedoptions$progressBarUpdater' is a function called in non-parallel simulations that can be used to update external progress bar.'advancedoptions$GUI' turns off some warning messages when in the GUI. If 'advancedoptions$save_simulated_responses = TRUE', the dataframe will have an attribute 'simulated_responses' that contains the simulated responses from the power evaluation. 'advancedoptions$ci_error_conf' will set the confidence level for power intervals, which are printed when 'detailedoutput = TRUE'.

`progress` Default 'TRUE'. Whether to include a progress bar.

`parallel` Default 'FALSE'. If 'TRUE', the power simulation will use all but one of the available cores. If the user wants to set the number of cores manually, they can do this by setting 'options("cores")' to the desired number (e.g. 'options("cores" = parallel::detectCores())'). NOTE: If you have installed BLAS libraries that include multicore support (e.g. Intel MKL that comes with Microsoft R Open), turning on parallel could result in reduced performance.

`parallel_pkgs` A vector of strings listing the external packages to be included in each parallel worker.

`...` Additional arguments.

Value

A data frame consisting of the parameters and their powers. The parameter estimates from the simulations are stored in the 'estimates' attribute.

Examples

```
#To demonstrate how a user can use their own libraries for Monte Carlo power generation,
#We will recreate eval_design_survival_mc using the eval_design_custom_mc framework.

#To begin, first let us generate the same design and random generation function shown in the
#eval_design_survival_mc examples:

basicdesign = expand.grid(a = c(-1, 1), b = c("a","b","c"))
design = gen_design(candidateset = basicdesign, model = ~a + b + a:b, trials = 100,
                         optimality = "D", repeats = 100)

#Random number generating function

rsurvival = function(X, b) {
 Y = rexp(n = nrow(X), rate = exp(-(X %*% b)))
 censored = Y > 1
 Y[censored] = 1
 return(survival::Surv(time = Y, event = !censored, type = "right"))
}

#We now need to tell the package how we want to fit our data,
#given the formula and the model matrix X (and, if needed, the list of contrasts).
#If the contrasts aren't required, "contrastslist" should be set to NULL.
#This should return some type of fit object.

fitsurv = function(formula, X, contrastslist = NULL) {
 return(survival::survreg(formula, data = X, dist = "exponential"))
}


#We now need to tell the package how to extract the p-values from the fit object returned
#from the fit function. This is how to extract the p-values from the survreg fit object:

pvalsurv = function(fit) {
 return(summary(fit)$table[, 4])
}

#And now we evaluate the design, passing the fitting function and p-value extracting function
#in along with the standard inputs for eval_design_mc.
#This has the exact same behavior as eval_design_survival_mc for the exponential distribution.
eval_design_custom_mc(design = design, model = ~a + b + a:b,
                      alpha = 0.05, nsim = 100,
                      fitfunction = fitsurv, pvalfunction = pvalsurv,
                      rfunction = rsurvival, effectsize = 1)
#We can also use skpr's framework for parallel computation to automatically parallelize this
#to speed up computation
## Not run: eval_design_custom_mc(design = design, model = ~a + b + a:b,
                          alpha = 0.05, nsim = 1000,
                          fitfunction = fitsurv, pvalfunction = pvalsurv,
                          rfunction = rsurvival, effectsize = 1,
                          parallel = TRUE, parallel_pkgs = "survival")
```

```
## End(Not run)
```

`eval_design_mc` *Monte Carlo Power Evaluation for Experimental Designs*

Description

Evaluates the power of an experimental design, given the run matrix and the statistical model to be fit to the data, using monte carlo simulation. Simulated data is fit using a generalized linear model and power is estimated by the fraction of times a parameter is significant. Returns a data frame of parameter powers.

Usage

```
eval_design_mc(
  design,
  model = NULL,
  alpha = 0.05,
  blocking = NULL,
  nsim = 1000,
  glmfamily = "gaussian",
  calceffect = TRUE,
  effect_anova = TRUE,
  varianceratios = NULL,
  rfunction = NULL,
  anticoef = NULL,
  firth = FALSE,
  effectsize = 2,
  contrasts = contr.sum,
  high_resolution_candidate_set = NA,
  moment_sample_density = 20,
  parallel = FALSE,
  adjust_alpha_inflation = FALSE,
  detailedoutput = FALSE,
  progress = TRUE,
  advancedoptions = NULL,
  ...
)
```

Arguments

`design`	The experimental design. Internally, `eval_design_mc` rescales each numeric column to the range [-1, 1].
`model`	The model used in evaluating the design. If this is missing and the design was generated with skpr, the generating model will be used. It can be a subset of the model used to generate the design, or include higher order effects not in the original design generation. It cannot include factors that are not present in the experimental design.

`alpha` Default '0.05'. The type-I error. p-values less than this will be counted as significant.

`blocking` Default 'NULL'. If 'TRUE', `eval_design_mc` will look at the rownames (or blocking columns) to determine blocking structure. Default FALSE.

`nsim` Default '1000'. The number of Monte Carlo simulations to perform.

`glmfamily` Default 'gaussian'. String indicating the family of distribution for the 'glm' function ("gaussian", "binomial", "poisson", or "exponential").

`calceffect` Default 'TRUE'. Whether to calculate effect power. This calculation is more expensive than parameter power, so turned off (if not needed) can greatly speed up calculation time.

`effect_anova` Default 'TRUE', whether to a Type-III Anova or a likelihood ratio test to calculate effect power. If 'TRUE', effect power will be calculated using a Type-III Anova (using the car package) and a Wald test. If 'FALSE', a likelihood ratio test (using a reduced model for each effect) will performed using the 'lmtest' package. If 'firth = TRUE', this will be set to 'FALSE' automatically.

`varianceratios` Default 'NULL'. The ratio of the whole plot variance to the run-to-run variance. If not specified during design generation, this will default to 1. For designs with more than one subplot this ratio can be a vector specifying the variance ratio for each subplot (comparing to the run-to-run variance). Otherwise, it will use a single value for all strata.

`rfunction` Default 'NULL'.Random number generator function for the response variable. Should be a function of the form f(X, b, delta), where X is the model matrix, b are the anticipated coefficients, and delta is a vector of blocking errors. Typically something like rnorm(nrow(X), X * b + delta, 1). You only need to specify this if you do not like the default behavior described below.

`anticoef` Default 'NULL'.The anticipated coefficients for calculating the power. If missing, coefficients will be automatically generated based on the `effectsize` argument.

`firth` Default 'FALSE'. Whether to apply the firth correction (via the 'mbest' package) to a logistic regression. This setting also automatically sets 'effect_lr = TRUE'.

`effectsize` Helper argument to generate anticipated coefficients. See details for more info. If you specify `anticoef`, `effectsize` will be ignored.

`contrasts` Default `contr.sum`. The contrasts to use for categorical factors. If the user has specified their own contrasts for a categorical factor using the contrasts function, those will be used. Otherwise, skpr will use contr.sum. If the user wants to set the number of cores manually, they can do this by setting 'options("cores")' to the desired number (e.g. 'options("cores" = parallel::detectCores())'). NOTE: If you have installed BLAS libraries that include multicore support (e.g. Intel MKL that comes with Microsoft R Open), turning on parallel could result in reduced performance.

`high_resolution_candidate_set`
Default 'NA'. If you have continuous numeric terms and disallowed combinations, the closed-form I-optimality value cannot be calculated and must be approximated by numeric integration. This requires sampling the allowed space

densely, but most candidate sets will provide a sparse sampling of allowable points. To work around this, skpr will generate a convex hull of the numeric terms for each unique combination of categorical factors to generate a dense sampling of the space and cache that value internally, but this is a slow calculation and does not support non-convex candidate sets. To speed up moment matrix calculation, pass a higher resolution version of your candidate set here with the disallowed combinations already applied. If you generated your design externally from skpr, there are disallowed combinations in your design, and need correct I-optimalituy values, you must pass your candidate set here.

`moment_sample_density`
: Default '20'. The density of points to sample when calculating the moment matrix to compute I-optimality. Only required if the design was generated outside of skpr and there are disallowed combinations.

`parallel`
: Default 'FALSE'. If 'TRUE', the Monte Carlo power calculation will use all but one of the available cores. If the user wants to set the number of cores manually, they can do this by setting 'options("cores")' to the desired number (e.g. 'options("cores" = parallel::detectCores())'). NOTE: If you have installed BLAS libraries that include multicore support (e.g. Intel MKL that comes with Microsoft R Open), turning on parallel could result in reduced performance.

`adjust_alpha_inflation`
: Default 'FALSE'. If 'TRUE', this will run the simulation twice: first to calculate the empirical distribution of p-values under the null hypothesis and find the true Type-I error cutoff that corresponds to the desired Type-I error rate, and then again given effect size to calculate power values.

`detailedoutput`
: Default 'FALSE'. If 'TRUE', return additional information about evaluation in results.

`progress`
: Default 'TRUE'. Whether to include a progress bar.

`advancedoptions`
: Default 'NULL'. Named list of advanced options. 'advancedoptions$anovatype' specifies the Anova type in the car package (default type 'III'), user can change to type 'II'). 'advancedoptions$anovatest' specifies the test statistic if the user does not want a 'Wald' test–other options are likelyhood-ratio 'LR' and F-test 'F'. 'advancedoptions$progressBarUpdater' is a function called in non-parallel simulations that can be used to update external progress bar.'advancedoptions$GUI' turns off some warning messages when in the GUI. If 'advancedoptions$save_simulated_responses = TRUE', the dataframe will have an attribute 'simulated_responses' that contains the simulated responses from the power evaluation. 'advancedoptions$ci_error_conf' will set the confidence level for power intervals, which are printed when 'detailedoutput = TRUE'.

`...`
: Additional arguments.

Details

Evaluates the power of a design with Monte Carlo simulation. Data is simulated and then fit with a generalized linear model, and the fraction of simulations in which a parameter is significant (its p-value, according to the fit function used, is less than the specified `alpha`) is the estimate of power for that parameter.

First, if `blocking = TURE`, the random noise from blocking is generated with `rnorm`. Each block gets a single sample of Gaussian random noise, with a variance as specified in `varianceratios`, and that sample is copied to each run in the block. Then, `rfunction` is called to generate a simulated response for each run of the design, and the data is fit using the appropriate fitting function. The functions used to simulate the data and fit it are determined by the `glmfamily` and `blocking` arguments as follows. Below, X is the model matrix, b is the anticipated coefficients, and d is a vector of blocking noise (if `blocking = FALSE` then d = 0):

glmfamily	blocking	rfunction	fit
"gaussian"	F	`rnorm(mean = X %*% b + d, sd = 1)`	`lm`
"gaussian"	T	`rnorm(mean = X %*% b + d, sd = 1)`	`lme4::lmer`
"binomial"	F	`rbinom(prob = 1/(1+exp(-(X %*% b + d))))`	`glm(family = "binomial")`
"binomial"	T	`rbinom(prob = 1/(1+exp(-(X %*% b + d))))`	`lme4::glmer(family = "binomial")`
"poisson"	F	`rpois(lambda = exp((X %*% b + d)))`	`glm(family = "poisson")`
"poisson"	T	`rpois(lambda = exp((X %*% b + d)))`	`lme4::glmer(family = "poisson")`
"exponential"	F	`rexp(rate = exp(-(X %*% b + d)))`	`glm(family = Gamma(link = "log"))`
"exponential"	T	`rexp(rate = exp(-(X %*% b + d)))`	`lme4:glmer(family = Gamma(link = "log"))`

Note that the exponential random generator uses the "rate" parameter, but `skpr` and `glm` use the mean value parameterization (= 1 / rate), hence the minus sign above. Also note that the gaussian model assumes a root-mean-square error of 1.

Power is dependent on the anticipated coefficients. You can specify those directly with the `anticoef` argument, or you can use the `effectsize` argument to specify an effect size and `skpr` will autogenerate them. You can provide either a length-1 or length-2 vector. If you provide a length-1 vector, the anticipated coefficients will be half of `effectsize`; this is equivalent to saying that the *linear predictor* (for a gaussian model, the mean response; for a binomial model, the log odds ratio; for an exponential model, the log of the mean value; for a poisson model, the log of the expected response) changes by `effectsize` when a continuous factor goes from its lowest level to its highest level. If you provide a length-2 vector, the anticipated coefficients will be set such that the *mean response* (for a gaussian model, the mean response; for a binomial model, the probability; for an exponential model, the mean response; for a poisson model, the expected response) changes from `effectsize[1]` to `effectsize[2]` when a factor goes from its lowest level to its highest level, assuming that the other factors are inactive (their x-values are zero).

The effect of a length-2 `effectsize` depends on the `glmfamily` argument as follows:

For `glmfamily = 'gaussian'`, the coefficients are set to `(effectsize[2] - effectsize[1]) / 2`.

For `glmfamily = 'binomial'`, the intercept will be `1/2 * log(effectsize[1] * effectsize[2] / (1 - effectsize[1]) / (1 - effectsize[2]))`, and the other coefficients will be `1/2 * log(effectsize[2] * (1 - effectsize[1]) / (1 - effectsize[2]) / effectsize[1])`.

For `glmfamily = 'exponential'` or `'poisson'`, the intercept will be `1 / 2 * (log(effectsize[2]) + log(effectsize[1]))`, and the other coefficients will be `1 / 2 * (log(effectsize[2]) - log(effectsize[1]))`.

Value

A data frame consisting of the parameters and their powers, with supplementary information stored in the data frame's attributes. The parameter estimates from the simulations are stored in the "estimates" attribute. The "model.matrix" attribute contains the model matrix that was used for power

evaluation, and also provides the encoding used for categorical factors. If you want to specify the anticipated coefficients manually, do so in the order the parameters appear in the model matrix.

Examples

```
#We first generate a full factorial design using expand.grid:
factorialcoffee = expand.grid(cost = c(-1, 1),
                        type = as.factor(c("Kona", "Colombian", "Ethiopian", "Sumatra")),
                              size = as.factor(c("Short", "Grande", "Venti")))
if(skpr:::run_documentation()) {
#And then generate the 21-run D-optimal design using gen_design.
designcoffee = gen_design(factorialcoffee,
                          model = ~cost + type + size, trials = 21, optimality = "D")
}
if(skpr:::run_documentation()) {
#To evaluate this design using a normal approximation, we just use eval_design
#(here using the default settings for contrasts, effectsize, and the anticipated coefficients):

eval_design(design = designcoffee, model = ~cost + type + size, 0.05)
}
if(skpr:::run_documentation()) {
#To evaluate this design with a Monte Carlo method, we enter the same information
#used in eval_design, with the addition of the number of simulations "nsim" and the distribution
#family used in fitting for the glm "glmfamily". For gaussian, binomial, exponential, and poisson
#families, a default random generating function (rfunction) will be supplied. If another glm
#family is used or the default random generating function is not adequate, a custom generating
#function can be supplied by the user. Like in `eval_design()`, if the model isn't entered, the
#model used in generating the design will be used.

eval_design_mc(designcoffee, nsim = 100, glmfamily = "gaussian")
}
if(skpr:::run_documentation()) {
#We can also add error bars on the Monte Carlo power values by setting
#`detailedoutput = TRUE` (which will print out other information as well).
#We can set the confidence via the `advancedoptions` argument.
eval_design_mc(designcoffee, nsim = 100, glmfamily = "gaussian",
               detailedoutput = TRUE, advancedoptions = list(ci_error_conf  = 0.8))
}
if(skpr:::run_documentation()) {
#We see here we generate approximately the same parameter powers as we do
#using the normal approximation in eval_design. Like eval_design, we can also change
#effectsize to produce a different signal-to-noise ratio:

eval_design_mc(design = designcoffee, nsim = 100,
                       glmfamily = "gaussian", effectsize = 1)
}
if(skpr:::run_documentation()) {
#Like eval_design, we can also evaluate the design with a different model than
#the one that generated the design.
eval_design_mc(design = designcoffee, model = ~cost + type, alpha = 0.05,
               nsim = 100, glmfamily = "gaussian")
}
if(skpr:::run_documentation()) {
```

```
#And here it is evaluated with additional interactions included:
eval_design_mc(design = designcoffee, model = ~cost + type + size + cost * type, 0.05,
              nsim = 100, glmfamily = "gaussian")
}
if(skpr:::run_documentation()) {
#We can also set "parallel = TRUE" to use all the cores available to speed up
#computation.
eval_design_mc(design = designcoffee, nsim = 10000,
                       glmfamily = "gaussian", parallel = TRUE)
}
if(skpr:::run_documentation()) {
#We can also evaluate split-plot designs. First, let us generate the split-plot design:

factorialcoffee2 = expand.grid(Temp = c(1, -1),
                               Store = as.factor(c("A", "B")),
                               cost = c(-1, 1),
                        type = as.factor(c("Kona", "Colombian", "Ethiopian", "Sumatra")),
                               size = as.factor(c("Short", "Grande", "Venti")))

vhtcdesign = gen_design(factorialcoffee2,
                       model = ~Store, trials = 6, varianceratio = 1)
htcdesign = gen_design(factorialcoffee2, model = ~Store + Temp, trials = 18,
                 splitplotdesign = vhtcdesign, blocksizes = rep(3, 6), varianceratio = 1)
splitplotdesign = gen_design(factorialcoffee2,
                            model = ~Store + Temp + cost + type + size, trials = 54,
                            splitplotdesign = htcdesign, blocksizes = rep(3, 18),
                            varianceratio = 1)

#Each block has an additional noise term associated with it in addition to the normal error
#term in the model. This is specified by a vector specifying the additional variance for
#each split-plot level. This is equivalent to specifying a variance ratio of one between
#the whole plots and the run-to-run variance for gaussian models.

#Evaluate the design. Note the decreased power for the blocking factors.
eval_design_mc(splitplotdesign, blocking = TRUE, nsim = 100,
                       glmfamily = "gaussian", varianceratios = c(1, 1, 1))
}
if(skpr:::run_documentation()) {
#We can also use this method to evaluate designs that cannot be easily
#evaluated using normal approximations. Here, we evaluate a design with a binomial response and see
#whether we can detect the difference between each factor changing whether an event occurs
#70% of the time or 90% of the time.

factorialbinom = expand.grid(a = c(-1, 1), b = c(-1, 1))
designbinom = gen_design(factorialbinom, model = ~a + b, trials = 90, optimality = "D")

eval_design_mc(designbinom, ~a + b, alpha = 0.2, nsim = 100, effectsize = c(0.7, 0.9),
              glmfamily = "binomial")
}
if(skpr:::run_documentation()) {
#We can also use this method to determine power for poisson response variables.
#Generate the design:
```

```
factorialpois = expand.grid(a = as.numeric(c(-1, 0, 1)), b = c(-1, 0, 1))
designpois = gen_design(factorialpois, ~a + b, trials = 70, optimality = "D")

#Evaluate the power:

eval_design_mc(designpois, ~a + b, 0.05, nsim = 100, glmfamily = "poisson",
               anticoef = log(c(0.2, 2, 2)))
}

#The coefficients above set the nominal value -- that is, the expected count
#when all inputs = 0 -- to 0.2 (from the intercept), and say that each factor
#changes this count by a factor of 4 (multiplied by 2 when x= +1, and divided by 2 when x = -1).
#Note the use of log() in the anticipated coefficients.
```

`eval_design_survival_mc`

Evaluate Power for Survival Design

Description

Evaluates power for an experimental design in which the response variable may be right- or left-censored. Power is evaluated with a Monte Carlo simulation, using the `survival` package and `survreg` to fit the data. Split-plot designs are not supported.

Usage

```
eval_design_survival_mc(
  design,
  model = NULL,
  alpha = 0.05,
  nsim = 1000,
  distribution = "gaussian",
  censorpoint = NA,
  censortype = "right",
  rfunctionsurv = NULL,
  anticoef = NULL,
  effectsize = 2,
  contrasts = contr.sum,
  parallel = FALSE,
  detailedoutput = FALSE,
  progress = TRUE,
  advancedoptions = NULL,
  ...
)
```

Arguments

`design`	The experimental design. Internally, all numeric columns will be rescaled to [-1, +1].

`model` The model used in evaluating the design. If this is missing and the design was generated with skpr, the generating model will be used. It can be a subset of the model used to generate the design, or include higher order effects not in the original design generation. It cannot include factors that are not present in the experimental design.

`alpha` Default '0.05'. The type-I error. p-values less than this will be counted as significant.

`nsim` The number of simulations. Default 1000.

`distribution` Distribution of survival function to use when fitting the data. Valid choices are described in the documentation for `survreg`. *Supported* options are "exponential", "lognormal", or "gaussian". Default "gaussian".

`censorpoint` The point after/before (for right-censored or left-censored data, respectively) which data should be labelled as censored. Default NA for no censoring. This argument is used only by the internal random number generators; if you supply your own function to the `rfunctionsurv` parameter, then this parameter will be ignored.

`censortype` The type of censoring (either "left" or "right"). Default "right".

`rfunctionsurv` Random number generator function. Should be a function of the form f(X, b), where X is the model matrix and b are the anticipated coefficients. This function should return a `Surv` object from the `survival` package. You do not need to provide this argument if `distribution` is one of the supported choices and you are satisfied with the default behavior described below.

`anticoef` The anticipated coefficients for calculating the power. If missing, coefficients will be automatically generated based on the `effectsize` argument.

`effectsize` Helper argument to generate anticipated coefficients. See details for more info. If you specify `anticoef`, `effectsize` will be ignored.

`contrasts` Default `contr.sum`. Function used to encode categorical variables in the model matrix. If the user has specified their own contrasts for a categorical factor using the contrasts function, those will be used. Otherwise, skpr will use contr.sum.

`parallel` Default 'FALSE'. If 'TRUE', the power simulation will use all but one of the available cores. If the user wants to set the number of cores manually, they can do this by setting 'options("cores")' to the desired number (e.g. 'options("cores" = parallel::detectCores())'). NOTE: If you have installed BLAS libraries that include multicore support (e.g. Intel MKL that comes with Microsoft R Open), turning on parallel could result in reduced performance.

`detailedoutput` Default 'FALSE'. If 'TRUE', return additional information about evaluation in results.

`progress` Default 'TRUE'. Whether to include a progress bar.

`advancedoptions`
Default 'NULL'. Named list of advanced options. Pass 'progressBarUpdater' to include function called in non-parallel simulations that can be used to update external progress bar. 'advancedoptions$ci_error_conf' will set the confidence level for power intervals, which are printed when 'detailedoutput = TRUE'.

`...` Any additional arguments to be passed into the `survreg` function during fitting.

Details

Evaluates the power of a design with Monte Carlo simulation. Data is simulated and then fit with a survival model (`survival::survreg`), and the fraction of simulations in which a parameter is significant (its p-value is less than the specified `alpha`) is the estimate of power for that parameter.

If not supplied by the user, `rfunctionsurv` will be generated based on the `distribution` argument as follows:

distribution	generating function
"gaussian"	`rnorm(mean = X %*% b, sd = 1)`
"exponential"	`rexp(rate = exp(-X %*% b))`
"lognormal"	`rlnorm(meanlog = X %*% b, sdlog = 1)`

In each case, if a simulated data point is past the censorpoint (greater than for right-censored, less than for left-censored) it is marked as censored. See the examples below for how to construct your own function.

Power is dependent on the anticipated coefficients. You can specify those directly with the `anticoef` argument, or you can use the `effectsize` argument to specify an effect size and `skpr` will autogenerate them. You can provide either a length-1 or length-2 vector. If you provide a length-1 vector, the anticipated coefficients will be half of `effectsize`; this is equivalent to saying that the *linear predictor* (for a gaussian model, the mean response; for an exponential model or lognormal model, the log of the mean value) changes by `effectsize` when a continuous factor goes from its lowest level to its highest level. If you provide a length-2 vector, the anticipated coefficients will be set such that the *mean response* changes from `effectsize[1]` to `effectsize[2]` when a factor goes from its lowest level to its highest level, assuming that the other factors are inactive (their x-values are zero).

The effect of a length-2 `effectsize` depends on the `distribution` argument as follows:

For `distribution = 'gaussian'`, the coefficients are set to `(effectsize[2] - effectsize[1]) / 2`.

For `distribution = 'exponential'` or `'lognormal'`, the intercept will be `1 / 2 * (log(effectsize[2]) + log(effectsize[1]))`, and the other coefficients will be `1 / 2 * (log(effectsize[2]) - log(effectsize[1]))`.

Value

A data frame consisting of the parameters and their powers. The parameter estimates from the simulations are stored in the 'estimates' attribute. The 'modelmatrix' attribute contains the model matrix and the encoding used for categorical factors. If you manually specify anticipated coefficients, do so in the order of the model matrix.

Examples

```
#These examples focus on the survival analysis case and assume familiarity
#with the basic functionality of eval_design_mc.

#We first generate a simple 2-level design using expand.grid:
basicdesign = expand.grid(a = c(-1, 1))
design = gen_design(candidateset = basicdesign, model = ~a, trials = 15)
```

```
#We can then evaluate the power of the design in the same way as eval_design_mc,
#now including the type of censoring (either right or left) and the point at which
#the data should be censored:

eval_design_survival_mc(design = design, model = ~a, alpha = 0.05,
                        nsim = 100, distribution = "exponential",
                        censorpoint = 5, censortype = "right")

#Built-in Monte Carlo random generating functions are included for the gaussian, exponential,
#and lognormal distributions.

#We can also evaluate different censored distributions by specifying a custom
#random generating function and changing the distribution argument.

rlognorm = function(X, b) {
  Y = rlnorm(n = nrow(X), meanlog = X %*% b, sdlog = 0.4)
  censored = Y > 1.2
  Y[censored] = 1.2
  return(survival::Surv(time = Y, event = !censored, type = "right"))
}

#Any additional arguments are passed into the survreg function call.  As an example, you
#might want to fix the "scale" argument to survreg, when fitting a lognormal:

eval_design_survival_mc(design = design, model = ~a, alpha = 0.2, nsim = 100,
                        distribution = "lognormal", rfunctionsurv = rlognorm,
                        anticoef = c(0.184, 0.101), scale = 0.4)
```

---

gen_design *Generate optimal experimental designs*

Description

Creates an experimental design given a model, desired number of runs, and a data frame of candidate test points. gen_design chooses points from the candidate set and returns a design that is optimal for the given statistical model.

Usage

```
gen_design(
  candidateset,
  model,
  trials,
  splitplotdesign = NULL,
  blocksizes = NULL,
  optimality = "D",
  augmentdesign = NULL,
  repeats = 20,
```

```
  custom_v = NULL,
  varianceratio = 1,
  contrast = contr.simplex,
  aliaspower = 2,
  minDopt = 0.8,
  k = NA,
  moment_sample_density = 20,
  high_resolution_candidate_set = NA,
  parallel = FALSE,
  progress = TRUE,
  add_blocking_columns = FALSE,
  randomized = TRUE,
  advancedoptions = NULL,
  timer = NULL
)
```

Arguments

`candidateset`	A data frame of candidate test points; each run of the optimal design will be chosen (with replacement) from this candidate set. Each row of the data frame is a candidate test point. Each row should be unique. Usually this is a full factorial test matrix generated for the factors in the model unless there are disallowed combinations of runs. Factors present in the candidate set but not present in the model are stripped out, and the duplicate entries in the candidate set are removed. Disallowed combinations can be specified by simply removing them from the candidate set. Disallowed combinations between a hard-to-change and an easy-to-change factor are detected by comparing an internal candidate set generated by the unique levels present in the candidate set and the split plot design. Those points are then excluded from the search. If a factor is continuous, its column should be type `numeric`. If a factor is categorical, its column should be type `factor` or `character`.
`model`	The statistical model used to generate the test design.
`trials`	The number of runs in the design.
`splitplotdesign`	If 'NULL', a fully randomized design is generated. If not NULL, a split-plot design is generated, and this argument specifies the design for all of the factors harder to change than the current set of factors. Each row corresponds to a block in which the harder to change factors will be held constant. Each row of `splitplotdesign` will be replicated as specified in `blocksizes`, and the optimal design is found for all of the factors given in the `model` argument, taking into consideration the fixed and replicated hard-to-change factors. If `blocksizes` is missing, 'gen_design' will attempt to allocate the runs in the most balanced design possible, given the number of blocks given in the argument 'splitplotdesign' and the total number of 'trials'.
`blocksizes`	Default 'NULL'. Specifies the block size(s) for design generation. If only one number is passed, 'gen_design()' will create blocks of the specified size, and if the total number of run specified in 'trials' is not divisible by the number,

'gen_design()' will attempt to allocate the runs in the most balanced design possible. If a list is passed, each entry in the list will specify an additional layer of blocking. If 'splitplotdesign' is not 'NULL', this argument specifies the number of subplots within each whole plot (each whole plot corresponding to a row in the 'splitplotdesign' data.frame).

`optimality` Default 'D'. The optimality criterion used in generating the design. Full list of supported criteria: "D", "I", "A", "ALIAS", "G", "T", "E", or "CUSTOM". If "CUSTOM", user must also define a function of the model matrix named 'customOpt' in their namespace that returns a single value, which the algorithm will attempt to optimize. For 'CUSTOM' optimality split-plot designs, the user must instead define 'customBlockedOpt', which should be a function of the model matrix and the variance-covariance matrix. For information on the algorithm behind Alias-optimal designs, see *Jones and Nachtsheim. "Efficient Designs With Minimal Aliasing." Technometrics, vol. 53, no. 1, 2011, pp. 62-71*.

`augmentdesign` Default NULL. A 'data.frame' of runs that are fixed during the optimal search process. The columns of 'augmentdesign' must match those of the candidate set. The search algorithm will search for the optimal 'trials' - 'nrow(augmentdesign)' remaining runs.

`repeats` Default '20'. The number of times to repeat the search for the best optimal design.

`custom_v` Default 'NULL'. The user can pass a custom variance-covariance matrix to be used during blocked design generation.

`varianceratio` Default '1'. The ratio between the block and run-to-run variance for a given stratum in a split plot/blocked design. This requires a design passed into 'splitplotdesign', so it will be overridden to '1' if no split plot design is entered.

`contrast` Default 'contr.simplex', an orthonormal sum contrast. Function used to generate the encoding for categorical variables.

`aliaspower` Default '2'. Degree of interactions to be used in calculating the alias matrix for alias optimal designs.

`minDopt` Default '0.8'. Minimum value for the D-Optimality of a design when searching for Alias-optimal designs.

`k` Default 'NA'. For D-optimal designs, this changes the search to a k-exchange algorithm *Johnson and Nachtsheim. "Some Guidelines for Constructing Exact D-Optimal Designs on Convex Design Spaces." Technometrics, vol. 25, 1983, pp. 271-277.* This exchanges only the k lowest variance runs in the design at each search iteration. Lower numbers can result in a faster search, but are less likely tofind an optimal design. Values of 'k >= n/4' have been shown empirically to generate similar designs to the full search. When 'k == trials', this results in the default modified Federov's algorithm. A 'k' of 1 is a form of Wynn's algorithm *Wynn. "Results in the Theory and Construction of D-Optimum Experimental Designs," Journal of the Royal Statistical Society, Ser. B,vol. 34, 1972, pp. 133-14*.

`moment_sample_density`
Default '20'. The density of points to sample when calculating the moment matrix to compute I-optimality if there are disallowed combinations. Otherwise, the closed-form moment matrix can be calculated.

`high_resolution_candidate_set`
: Default 'NA'. If you have continuous numeric terms and disallowed combinations, the closed-form I-optimality value cannot be calculated and must be approximated by numeric integration. This requires sampling the allowed space densely, but most candidate sets will provide a sparse sampling of allowable points. To work around this, skpr will generate a convex hull of the numeric terms for each unique combination of categorical factors to generate a dense sampling of the space and cache that value internally, but this is a slow calculation and does not support non-convex candidate sets. To speed up moment matrix calculation, pass a higher resolution version of your candidate set here with the disallowed combinations already applied.

`parallel`
: Default 'FALSE'. If 'TRUE', the optimal design search will use all but one of the available cores. This can lead to a substantial speed-up in the search for complex designs. If the user wants to set the number of cores manually, they can do this by setting 'options("cores")' to the desired number (e.g. 'options("cores" = parallel::detectCores())'). NOTE: If you have installed BLAS libraries that include multicore support (e.g. Intel MKL that comes with Microsoft R Open), turning on parallel could result in reduced performance.

`progress`
: Default 'TRUE'. Whether to include a progress bar.

`add_blocking_columns`
: Default 'FALSE'. The blocking structure of the design will be indicated in the row names of the returned design. If 'TRUE', the design also will have extra columns to indicate the blocking structure. If no blocking is detected, no columns will be added.

`randomized`
: Default 'TRUE', due to the intrinsic randomization of the design search algorithm. If 'FALSE', the randomized design will be re-ordered from left to right.

`advancedoptions`
: Default 'NULL'. An named list for advanced users who want to adjust the optimal design algorithm parameters. Advanced option names are 'design_search_tolerance' (the smallest fractional increase below which the design search terminates), 'alias_tie_power' (the degree of the aliasing matrix when calculating optimality tie-breakers), 'alias_tie_tolerance' (the smallest absolute difference in the optimality criterion where designs are considered equal before considering the aliasing structure), 'alias_compare" (which if set to FALSE turns off alias tie breaking completely), 'aliasmodel' (provided if the user does not want to calculate Alias-optimality using all 'aliaspower' interaction terms), and 'progressBarUpdater" (a function called in non-parallel optimal searches that can be used to update an external progress bar). Finally, there's 'g_efficiency_method', which sets the method used to calculate G-efficiency (default is "random" for a random Monte Carlo sampling of the design space, "optim" for to use simulated annealing, or "custom" to explicitly define the points in the design space, which is the fastest method and the only way to calculate prediction variance with disallowed combinations). With this, there's also 'g_efficiency_samples', which specifies the number of random samples (default 1000 if 'g_efficiency_method = "random"'), attempts at simulated annealing (default 1 if 'g_efficiency_method = "optim"'), or a data.frame defining the exact points of the design space if 'g_efficiency_method = "custom"'.

`timer`
: Deprecated: Use 'progress' instead.

Details

Split-plot designs can be generated with repeated applications of gen_design; see examples for details.

Value

A data frame containing the run matrix for the optimal design. The returned data frame contains supplementary information in its attributes, which can be accessed with the ‘get_attributes()‘ and ‘get_optimality()‘ functions.

Examples

```
#Generate the basic factorial candidate set with expand.grid.
#Generating a basic 2 factor candidate set:
basic_candidates = expand.grid(x1 = c(-1, 1), x2 = c(-1, 1))

#This candidate set is used as an input in the optimal design generation for a
#D-optimal design with 11 runs.
design = gen_design(candidateset = basic_candidates, model = ~x1 + x2, trials = 11)

#We can also use the dot formula to automatically use all of the terms in the model:
design = gen_design(candidateset = basic_candidates, model = ~., trials = 11)

#Here we add categorical factors, specified by using "as.factor" in expand.grid:
categorical_candidates = expand.grid(a = c(-1, 1),
                                     b = as.factor(c("A", "B")),
                                     c = as.factor(c("High", "Med", "Low")))

#This candidate set is used as an input in the optimal design generation.
design2 = gen_design(candidateset = categorical_candidates, model = ~a + b + c, trials = 19)

#We can also increase the number of times the algorithm repeats
#the search to increase the probability that the globally optimal design was found.
design2 = gen_design(candidateset = categorical_candidates,
                     model = ~a + b + c, trials = 19, repeats = 100)

#We can perform a k-exchange algorithm instead of a full search to help speed up
#the search process, although this can lead to less optimal designs. Here, we only
#exchange the 10 lowest variance runs in each search iteration.
if(skpr:::run_documentation()) {
design_k = gen_design(candidateset = categorical_candidates,
                     model = ~a + b + c, trials = 19, repeats = 100, k = 10)
}

#To speed up the design search, you can turn on multicore support with the parallel option.
#You can also customize the number of cores used by setting the cores option. By default,
#all cores are used.
if(skpr:::run_documentation()) {
options(cores = 2)
design2 = gen_design(categorical_candidates,
                     model = ~a + b + c, trials = 19, repeats = 1000, parallel = TRUE)
}
```

```
#You can also use a higher order model when generating the design:
design2 = gen_design(categorical_candidates,
                     model = ~a + b + c + a * b * c, trials = 12, repeats = 10)

#To evaluate a response surface design, include center points
#in the candidate set and include quadratic effects (but not for the categorical factors).

quad_candidates = expand.grid(a = c(1, 0, -1), b = c(-1, 0, 1), c = c("A", "B", "C"))

gen_design(quad_candidates, ~a + b + I(a^2) + I(b^2) + a * b * c, 20)

#The optimality criterion can also be changed:
gen_design(quad_candidates, ~a + b + I(a^2) + I(b^2) + a * b * c, 20,
          optimality = "I", repeats = 10)
gen_design(quad_candidates, ~a + b + I(a^2) + I(b^2) + a * b * c, 20,
          optimality = "A", repeats = 10)

#A blocked design can be generated by specifying the `blocksizes` argument. Passing a single
#number will create designs with blocks of that size, while passing multiple values in a list
#will specify multiple layers of blocking.

#Specify a single layer
gen_design(quad_candidates, ~a + b + c, 21, blocksizes=3, add_blocking_column=TRUE)

#Manually specify the block sizes for a single layer, must add to `trials``
gen_design(quad_candidates, ~a + b + c, 21, blocksizes=c(4,3,2,3,3,3,3),
          add_blocking_column=TRUE)

#Multiple layers of blocking
gen_design(quad_candidates, ~a + b + c, 21, blocksizes=list(7,3),
          add_blocking_column=TRUE)

#Multiple layers of blocking, specified individually
gen_design(quad_candidates, ~a + b + c, 21, blocksizes=list(7,c(4,3,2,3,3,3,3)),
          add_blocking_column=TRUE)

#A split-plot design can be generated by first generating an optimal blocking design using the
#hard-to-change factors and then using that as the input for the split-plot design.
#This generates an optimal subplot design that accounts for the existing split-plot settings.

splitplotcandidateset = expand.grid(Altitude = c(-1, 1),
                                    Range = as.factor(c("Close", "Medium", "Far")),
                                    Power = c(1, -1))
hardtochangedesign = gen_design(splitplotcandidateset, model = ~Altitude,
                                trials = 11, repeats = 10)

#Now we can use the D-optimal blocked design as an input to our full design.

#Here, we add the easy to change factors from the candidate set to the model,
#and input the hard-to-change design along with the new number of trials. `gen_design` will
#automatically allocate the runs in the blocks in the most balanced way possible.
```

```
designsplitplot = gen_design(splitplotcandidateset, ~Altitude + Range + Power, trials = 33,
                             splitplotdesign = hardtochangedesign, repeats = 10)

#If we want to allocate the blocks manually, we can do that with the argument `blocksizes`. This
#vector must sum to the number of `trials` specified.

#Putting this all together:
designsplitplot = gen_design(splitplotcandidateset, ~Altitude + Range + Power, trials = 33,
                             splitplotdesign = hardtochangedesign,
                            blocksizes = c(4, 2, 3, 4, 2, 3, 4, 2, 3, 4, 2), repeats = 10)

#The split-plot structure is encoded into the row names, with a period
#demarcating the blocking level. This process can be repeated for arbitrary
#levels of blocking (i.e. a split-plot design can be entered in as the hard-to-change
#to produce a split-split-plot design, which can be passed as another
#hard-to-change design to produce a split-split-split plot design, etc).
#In the following, note that the model builds up as we build up split plot strata.

splitplotcandidateset2 = expand.grid(Location = as.factor(c("East", "West")),
                                     Climate = as.factor(c("Dry", "Wet", "Arid")),
                                     Vineyard = as.factor(c("A", "B", "C", "D")),
                                     Age = c(1, -1))
#6 blocks of Location:
temp = gen_design(splitplotcandidateset2, ~Location, trials = 6, varianceratio = 2, repeats = 10)

#Each Location block has 2 blocks of Climate:
temp = gen_design(splitplotcandidateset2, ~Location + Climate,
                  trials = 12, splitplotdesign = temp, blocksizes = 2,
                  varianceratio = 1, repeats = 10)

#Each Climate block has 4 blocks of Vineyard:
temp = gen_design(splitplotcandidateset2, ~Location + Climate + Vineyard,
                  trials = 48, splitplotdesign = temp, blocksizes = 4,
                  varianceratio = 1, repeats = 10)

#Each Vineyard block has 4 runs with different Age:
if(skpr:::run_documentation()) {
splitsplitsplitplotdesign = gen_design(splitplotcandidateset2, ~Location + Climate + Vineyard + Age,
                                    trials = 192, splitplotdesign = temp, blocksizes = 4,
                                       varianceratio = 1, add_blocking_columns = TRUE)
}
#gen_design also supports user-defined optimality criterion. The user defines a function
#of the model matrix named customOpt, and gen_design will attempt to generate a design
#that maximizes that function. This function needs to be in the global environment, and be
#named either customOpt or customBlockedOpt, depending on whether a split-plot design is being
#generated. customBlockedOpt should be a function of the model matrix as well as the
#variance-covariance matrix, vInv. Due to the underlying C + + code having to call back to the R
#environment repeatedly, this criterion will be significantly slower than the built-in algorithms.
#It does, however, offer the user a great deal of flexibility in generating their designs.

#We are going to write our own D-optimal search algorithm using base R functions. Here, write
#a function that calculates the determinant of the information matrix. gen_design will search
#for a design that maximizes this function.
```

```
customOpt = function(currentDesign) {
 return(det(t(currentDesign) %*% currentDesign))
}

#Generate the whole plots for our split-plot design, using the custom criterion.

candlistcustom = expand.grid(Altitude = c(10000, 20000),
                             Range = as.factor(c("Close", "Medium", "Far")),
                             Power = c(50, 100))
htcdesign = gen_design(candlistcustom, model = ~Altitude + Range,
                       trials = 11, optimality = "CUSTOM", repeats = 10)

#Now define a function that is a function of both the model matrix,
#as well as the variance-covariance matrix vInv. This takes the blocking structure into account
#when calculating our determinant.

customBlockedOpt = function(currentDesign, vInv) {
 return(det(t(currentDesign) %*% vInv %*% currentDesign))
}

#And finally, calculate the design. This (likely) results in the same design had we chosen the
#"D" criterion.

design = gen_design(candlistcustom,
                    ~Altitude + Range + Power, trials = 33,
                    splitplotdesign = htcdesign, blocksizes = 3,
                    optimality = "CUSTOM", repeats = 10)

#gen_design can also augment an existing design. Input a dataframe of pre-existing runs
#to the `augmentdesign` argument. Those runs in the new design will be fixed, and gen_design
#will perform a search for the remaining `trials - nrow(augmentdesign)` runs.

candidateset = expand.grid(height = c(10, 20), weight = c(45, 55, 65), range = c(1, 2, 3))

design_to_augment = gen_design(candidateset, ~height + weight + range, 5)

#As long as the columns in the augmented design match the columns in the candidate set,
#this design can be augmented.

augmented_design = gen_design(candidateset,
                          ~height + weight + range, 16, augmentdesign = design_to_augment)

#A design's diagnostics can be accessed via the `get_optimality()` function:

get_optimality(augmented_design)

#And design attributes can be accessed with the `get_attribute()` function:

get_attribute(design)

#A correlation color map can be produced by calling the plot_correlation command with the output
#of gen_design()
```

```
if(skpr:::run_documentation()) {
plot_correlations(design2)
}

#A fraction of design space plot can be produced by calling the plot_fds command
if(skpr:::run_documentation()) {
plot_fds(design2)
}

#Evaluating the design for power can be done with eval_design, eval_design_mc (Monte Carlo)
#eval_design_survival_mc (Monte Carlo survival analysis), and
#eval_design_custom_mc (Custom Library Monte Carlo)
```

`get_attribute` *Get attribute values*

Description

Returns one or more of underlying attributes used in design generation/evaluation

Usage

```
get_attribute(output, attr = NULL, round = TRUE)
```

Arguments

`output`	The output of either ‘gen_design()‘ or ‘eval_design()‘/‘eval_design_mc()“
`attr`	Default ‘NULL‘. Return just the specific value requested. Potential values are ‘model.matrix‘ for model used, ‘moments.matrix‘, ‘variance.matrix‘, ‘alias.matrix‘, ‘correlation.matrix‘, and ‘model‘ for the model used in the evaluation/generation of the design.
`round`	Default ‘TRUE‘. Rounds off values smaller than the magnitude ‘1e-15“ in the ‘correlation.matrix‘ and ‘alias.matrix‘ matrix attributes.

Value

A list of attributes.

Examples

```
# We can extract the attributes of a design from either the output of `gen_design()`
# or the output of `eval_design()`

factorialcoffee = expand.grid(cost = c(1, 2),
                        type = as.factor(c("Kona", "Colombian", "Ethiopian", "Sumatra")),
                             size = as.factor(c("Short", "Grande", "Venti")))

designcoffee = gen_design(factorialcoffee, ~cost + size + type, trials = 29,
```

```
                          optimality = "D", repeats = 100)

#Extract a list of all attributes
get_attribute(designcoffee)

#Get just one attribute
get_attribute(designcoffee,"model.matrix")

# Extract from `eval_design()` output
power_output = eval_design(designcoffee, model = ~cost + size + type,
                           alpha = 0.05, detailedoutput = TRUE)

get_attribute(power_output,"correlation.matrix")
```

`get_optimality` *Get optimality values*

Description

Returns a list of optimality values (or one value in particular).

Note: The choice of contrast will effect the 'G' efficiency value, and 'gen_design()' and 'eval_design()' by default set different contrasts ('contr.simplex' vs 'contr.sum').

Usage

```
get_optimality(output, optimality = NULL, calc_g = FALSE)
```

Arguments

`output`	The output of either gen_design or eval_design/eval_design_mc.
`optimality`	Default 'NULL'. Return just the specific optimality requested.
`calc_g`	Default 'FALSE'. Whether to calculate the g-efficiency.

Value

A dataframe of optimality conditions. 'D', 'A', and 'G' are efficiencies (value is out of 100). 'T' is the trace of the information matrix, 'E' is the minimum eigenvalue of the information matrix, 'I' is the average prediction variance, and 'Alias' is the trace of the alias matrix.

Examples

```
# We can extract the optimality of a design from either the output of `gen_design()`
# or the output of `eval_design()`

factorialcoffee = expand.grid(cost = c(1, 2),
                        type = as.factor(c("Kona", "Colombian", "Ethiopian", "Sumatra")),
                             size = as.factor(c("Short", "Grande", "Venti")))
```

```
designcoffee = gen_design(factorialcoffee, ~cost + size + type, trials = 29,
                          optimality = "D", repeats = 100)

#Extract a list of all attributes
get_optimality(designcoffee)

#Get just one attribute
get_optimality(designcoffee,"D")

# Extract from `eval_design()` output
power_output = eval_design(designcoffee, model = ~cost + size + type,
                           alpha = 0.05, detailedoutput = TRUE)

get_optimality(power_output)
```

`get_power_curve_output` *Get Power Curve Warnings and Errors*

Description

Gets the warnings and errors from 'calculate_power_curves()' output.

Usage

```
get_power_curve_output(power_curve)
```

Arguments

`power_curve`	The output from 'calculate_power_curves()'

Value

A list of data.frames containing warning/error information

Examples

```
#Generate sample
if(skpr:::run_documentation()) {
calculate_power_curves(trials=seq(50,150,by=20),
                       candidateset = expand.grid(x=c(-1,1),y=c(-1,1)),
                       model = ~.,
                       effectsize = list(c(0.5,0.9),c(0.6,0.9)),
                       eval_function = eval_design_mc,
                       eval_args = list(nsim = 100, glmfamily = "binomial"))
}
```

plot_correlations *Plots design diagnostics*

Description

Plots design diagnostics

Usage

```
plot_correlations(
  genoutput,
  model = NULL,
  customcolors = NULL,
  pow = 2,
  custompar = NULL,
  standardize = TRUE,
  plot = TRUE
)
```

Arguments

genoutput	The output of either gen_design or eval_design/eval_design_mc
model	Default 'NULL'. Defaults to the model used in generating/evaluating the design, augmented with 2-factor interactions. If specified, it will override the default model used to generate/evaluate the design.
customcolors	A vector of colors for customizing the appearance of the colormap
pow	Default 2. The interaction level that the correlation map is showing.
custompar	Default NULL. Custom parameters to pass to the 'par' function for base R plotting.
standardize	Default 'TRUE'. Whether to standardize (scale to -1 and 1 and center) the continuous numeric columns. Not standardizing the numeric columns can increase multi-collinearity (predictors that are correlated with other predictors in the model).
plot	Default 'TRUE'. If 'FALSE', this will return the correlation matrix.

Value

Silently returns the correlation matrix with the proper row and column names.

Examples

```
#We can pass either the output of gen_design or eval_design to plot_correlations
#in order to obtain the correlation map. Passing the output of eval_design is useful
#if you want to plot the correlation map from an externally generated design.

#First generate the design:
```

```
candidatelist = expand.grid(cost = c(15000, 20000), year = c("2001", "2002", "2003", "2004"),
                            type = c("SUV", "Sedan", "Hybrid"))
cardesign = gen_design(candidatelist, ~(cost+type+year)^2, 30)
plot_correlations(cardesign)

#We can also increase the level of interactions that are shown by default.

plot_correlations(cardesign, pow = 3)

#You can also pass in a custom color map.
plot_correlations(cardesign, customcolors = c("blue", "grey", "red"))
plot_correlations(cardesign, customcolors = c("blue", "green", "yellow", "orange", "red"))
```

`plot_fds` *Fraction of Design Space Plot*

Description

Creates a fraction of design space plot

Usage

```
plot_fds(
  genoutput,
  model = NULL,
  continuouslength = 1001,
  plot = TRUE,
  sample_size = 10000,
  yaxis_max = NULL,
  description = "Fraction of Design Space"
)
```

Arguments

`genoutput`	The design, or the output of the power evaluation functions. This can also be a list of several designs, which will result in all of them being plotted in a row (for easy comparison).
`model`	Default 'NULL'. The model, if 'NULL' it defaults to the model used in 'eval_design' or 'gen_design'.
`continuouslength`	Default '11'. The precision of the continuous variables. Decrease for faster (but less precise) plotting.
`plot`	Default 'TRUE'. Whether to plot the FDS, or just calculate the cumulative distribution function.
`sample_size`	Default '10000'. Number of samples to take of the design space.
`yaxis_max`	Default 'NULL'. Manually set the maximum value of the prediction variance.

description Default ‘Fraction of Design Space‘. The description to add to the plot. If a vector and multiple designs passed to genoutput, it will be the description for each plot.

Value

Plots design diagnostics, and invisibly returns the vector of values representing the fraction of design space plot. If multiple designs are passed, this will return a list of all FDS vectors.

Examples

```
#We can pass either the output of gen_design or eval_design to plot_correlations
#in order to obtain the correlation map. Passing the output of eval_design is useful
#if you want to plot the correlation map from an externally generated design.

#First generate the design:

candidatelist = expand.grid(X1 = c(1, -1), X2 = c(1, -1))

design = gen_design(candidatelist, ~(X1 + X2), 15)

plot_fds(design)
```

---

print.skpr_eval_output

Print evaluation information

Description

Prints design evaluation information below the data.frame of power values

Note: If options("skpr.ANSI") is ‘NULL‘ or ‘TRUE‘, ANSI codes will be used during printing to prettify the output. If this is ‘FALSE‘, only ASCII will be used.

Usage

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

Arguments

x	The x of the evaluation functions in skpr
...	Additional arguments.

Examples

```
#Generate/evaluate a design and print its information
factorialcoffee = expand.grid(cost = c(1, 2),
                        type = as.factor(c("Kona", "Colombian", "Ethiopian", "Sumatra")),
                              size = as.factor(c("Short", "Grande", "Venti")))

designcoffee = gen_design(factorialcoffee,
                       ~cost + size + type, trials = 29, optimality = "D", repeats = 100)

eval_design(designcoffee)
```

---

print.skpr_power_curve_output

Print evaluation information

Description

Prints design evaluation information below the data.frame of power values

Note: If options("skpr.ANSI") is 'NULL' or 'TRUE', ANSI codes will be used during printing to prettify the output. If this is 'FALSE', only ASCII will be used.

Usage

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

Arguments

x	The x of the evaluation functions in skpr
...	Additional arguments.

Examples

```
#Generate/evaluate a design and print its information
factorialcoffee = expand.grid(cost = c(1, 2),
                              type = as.factor(c("Kona",
                                                 "Colombian",
                                                 "Ethiopian",
                                                 "Sumatra")),
                              size = as.factor(c("Short",
                                                 "Grande",
                                                 "Venti")))

coffee_curves = calculate_power_curves(candidateset = factorialcoffee,
                                       model = ~(cost + size + type)^2,
                                       trials = 30:40, plot_results = FALSE)
coffee_curves
```

skprGUI *Graphical User Interface for skpr*

Description

skprGUI provides a graphical user interface to skpr, within R Studio.

Usage

```
skprGUI(
  browser = FALSE,
  return_app = FALSE,
  multiuser = FALSE,
  progress = TRUE
)
```

Arguments

browser	Default 'FALSE'. Whether to open the application in an external browser.
return_app	Default 'FALSE'. If 'TRUE', this will return the shinyApp object.
multiuser	Default 'FALSE'. If 'TRUE', this will turn off and disable multicore functionality and enable non-blocking operation.
progress	Default 'TRUE'. Whether to include a progress bar in the application. Note: if 'multiuser = TRUE', progress bars are turned on by default.

Examples

```
#Type `skprGUI()` to begin
```

%>% *re-export magrittr pipe operator*

Description

re-export magrittr pipe operator

Index

%>%, 37

calculate_power_curves, 2
contr.simplex, 4

eval_design, 5
eval_design_custom_mc, 9
eval_design_mc, 13
eval_design_survival_mc, 19

gen_design, 22
get_attribute, 30
get_optimality, 31
get_power_curve_output, 32

plot_correlations, 33
plot_fds, 34
print.skpr_eval_output, 35
print.skpr_power_curve_output, 36

skprGUI, 37