Package 'skpr'

November 13, 2018

Title Design of Experiments Suite: Generate and Evaluate Optimal Designs

Version 0.57.0

Description
Generates and evaluates D, I, A, Alias, E, T, and G optimal designs. Supports generation and evaluation of 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.

Date 2018-11-07

Copyright Institute for Defense Analyses

Depends R (>= 3.0.2), shiny

License GPL-3

LazyData true

RoxygenNote 6.1.0

Imports utils, iterators, stats, nlme, lme4, Rcpp (>= 0.11.0), rintrojs, shinythemes, foreach, doParallel, survival, knitr, kableExtra, doRNG, future, promises, shinyjs, car, viridis, magrittr

LinkingTo Rcpp, RcppEigen

Suggests testthat

Encoding UTF-8

URL https://github.com/tylermorganwall/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 2018-11-13 19:20:03 UTC

R topics documented:

`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.
`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, alpha, blocking = FALSE, anticoef = NULL,
  effectsize = 2, varianceratios = 1, contrasts = contr.sum,
  conservative = FALSE, detailedoutput = FALSE, ...)
```

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. 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`	The specified type-I error.
`blocking`	Default FALSE. If TRUE, `eval_design` will look at the rownames to determine blocking structure.
`anticoef`	The anticipated coefficients for calculating the power. If missing, coefficients will be automatically generated based on the `effectsize` argument.
`effectsize`	The signal-to-noise ratio. Default 2. 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 1. The ratio of the whole plot variance to the run-to-run variance. For designs with more than one subplot this ratio can be a vector specifying the variance ratio for each subplot. 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 level in each categorical factor's anticipated coefficients to zero. Default FALSE.
detailedoutput	If TRUE, return additional information about evaluation in results. Default FALSE.
...	Additional arguments.

Details

This function evaluates the power of experimental designs.

Power is calculated under a linear regression framework: you intend to fit a linear model to the data, of the form

$y = X\beta + \epsilon$, (plus blocking terms, if applicable)

where y is the vector of experimental responses, X is the model matrix, β is the vector of model coefficients, and ϵ is the statistical noise. `eval_design` assumes that ϵ is normally distributed with zero mean and unit variance (root-mean-square error is 1), and calculates both parameter power and effect power. Parameter power is the probability of rejecting the hypothesis $\beta_i = 0$, where β_i is a single parameter in the model, while effect power is the probability of rejecting the hypothesis $\beta_i = 0$, where β_i is the set of all parameters associated with the effect in question. The two powers are equivalent for continuous factors and two-level categorical factors, but they can be different for categorical factors with three or more levels.

When using `conservative = TRUE`, `eval_design` first evaluates the power with default 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.

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)

#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).
#(Setting detailedoutput = TRUE provides information on the anticipated
#coefficients that were used:)
eval_design(designcoffee, model = ~cost + size + type, alpha = 0.05, detailedoutput = TRUE)

#Evaluate the design, with conservative anticipated coefficients:
eval_design(designcoffee, model = ~cost + size + type, alpha = 0.05, detailedoutput = TRUE,
            conservative = TRUE)

#which is the same as the following, but now explicitly entering the coefficients:
eval_design(designcoffee, model = ~cost + size + type, alpha = 0.05,
            anticoef = c(1, 1, 1, 0, 0, 1, 0), detailedoutput = TRUE)


#If the defaults do not suit you, enter the anticipated coefficients in manually.
eval_design(designcoffee,
           model = ~cost + size + type, alpha = 0.05, anticoef = c(1, 1, 0, 0, 1, 0, 1))

#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, alpha = 0.05)

#Split plot designs can also be evaluated by setting the blocking parameter as TRUE.

#Generating 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, splitplotsizes = 3)
```

```
#Evaluating design
eval_design(coffeefinaldesign, ~cost + size + type + caffeine, 0.2, blocking = TRUE)

#We can also evaluate the design with a custom ratio between the whole plot error to
#the run-to-run error.
eval_design(coffeefinaldesign, ~caffeine + cost + size + type + caffeine, 0.2, blocking = TRUE,
            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, the rownames will be as follows:

manualrownames = paste(c(1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4), rep(c(1, 2, 3), 4), sep = ".")

#If we wanted to add this blocking structure to the design coffeeblockdesign, we would
#simply set the rownames to this character vector:

rownames(coffeeblockdesign) = manualrownames

#Deeper levels of blocking can be specified with additional periods.
```

`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, alpha, nsim, rfunction, fitfunction,
  pvalfunction, anticoef, effectsize = 2, contrasts = contr.sum,
  coef_function = coef, parameternames = NULL, parallel = FALSE,
  parallelpackages = NULL, ...)
```

Arguments

`design`	The experimental design. Internally, `eval_design_custom_mc` rescales each numeric column to the range [-1, 1].
`model`	The statistical model used to fit the data.
`alpha`	The type-I error.
`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.
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).
parallel	If TRUE, uses all cores available to speed up computation of power. Default FALSE.
parallelpackages	A vector of strings listing the external packages to be input into the parallel package.
...	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))
design = gen_design(candidateset = basicdesign, model = ~a, 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.

d = eval_design_custom_mc(design = design, model = ~a,
                          alpha = 0.05, nsim = 100,
                          fitfunction = fitsurv, pvalfunction = pvalsurv,
                          rfunction = rsurvival, effectsize = 1)

#This has the exact same behavior as eval_design_survival_mc for the exponential distribution.
```

`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, alpha, blocking = FALSE, nsim = 1000,
  glmfamily = "gaussian", calceffect = TRUE, varianceratios = NULL,
  rfunction = NULL, anticoef = NULL, effectsize = 2,
  contrasts = contr.sum, parallel = FALSE, detailedoutput = FALSE,
  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. 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	The type-I error. p-values less than this will be counted as significant.
blocking	If TRUE, eval_design_mc will look at the rownames to determine blocking structure. Default FALSE.
nsim	The number of simulations to perform.
glmfamily	String indicating the family of distribution for the glm function ("gaussian", "binomial", "poisson", or "exponential").
calceffect	Default 'TRUE'. 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.
varianceratios	Default '1'. The ratio of the whole plot variance to the run-to-run variance. For designs with more than one subplot this ratio can be a vector specifying the variance ratio for each subplot. Otherwise, it will use a single value for all strata.
rfunction	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	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. 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.
parallel	Default FALSE. If TRUE, uses all cores available to speed up computation. WARNING: This can slow down computation if nonparallel time to complete the computation is less than a few seconds.
detailedoutput	If TRUE, return additional information about evaluation in results.
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.
...	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	
"gaussian"	F	`rnorm(mean = X %*% b + d, sd = 1)`	
"gaussian"	T	`rnorm(mean = X %*% b + d, sd = 1)`	`lme4::`
"binomial"	F	`rbinom(prob = 1/(1+exp(-(X %*% b + d))))`	`glm(family = "binomi`
"binomial"	T	`rbinom(prob = 1/(1+exp(-(X %*% b + d))))`	`lme4::glmer(family = "binomi`
"poisson"	F	`rpois(lambda = exp((X %*% b + d)))`	`glm(family = "poiss`
"poisson"	T	`rpois(lambda = exp((X %*% b + d)))`	`lme4::glmer(family = "poiss`
"exponential"	F	`rexp(rate = exp(-(X %*% b + d)))`	`glm(family = Gamma(link = "lo`
"exponential"	T	`rexp(rate = exp(-(X %*% b + d)))`	`lme4:glmer(family = Gamma(link = "lo`

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`
and the other coefficients will be `1/2 * log(effectsize[2] * (1 - effectsize[1]) / (1 - effectsize[2]) / effect`

For `glmfamily = 'exponential'` or `'poisson'`, the intercept will be `1 / 2 * (log(effectsize[2]) + log(effectsize`
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 "modelmatrix" 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")))

#And then generate the 21-run D-optimal design using gen_design.

designcoffee = gen_design(factorialcoffee,
                         model = ~cost + type + size, trials = 21, optimality = "D")

#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)

#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.

## Not run: eval_design_mc(designcoffee, model = ~cost + type + size, alpha = 0.05, nsim = 100,
                        glmfamily = "gaussian")
## End(Not run)

#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:

## Not run: eval_design_mc(design = designcoffee, model = ~cost + type + size, alpha = 0.05,
              nsim = 100, glmfamily = "gaussian", effectsize = 1)
## End(Not run)

#Like eval_design, we can also evaluate the design with a different model than
#the one that generated the design.
## Not run: eval_design_mc(design = designcoffee, model = ~cost + type, alpha = 0.05,
              nsim = 100, glmfamily = "gaussian")
## End(Not run)


#And here it is evaluated with interactions included:
## Not run: eval_design_mc(design = designcoffee, model = ~cost + type + size + cost * type, 0.05,
```

```
              nsim = 100, glmfamily = "gaussian")
## End(Not run)

#We can also set "parallel = TRUE" to use all the cores available to speed up
#computation.
## Not run: eval_design_mc(design = designcoffee, model = ~cost + type + size, 0.05,
              nsim = 10000, glmfamily = "gaussian", parallel = TRUE)
## End(Not run)

#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, splitplotsizes = rep(3, 6), varianceratio = 1)
splitplotdesign = gen_design(factorialcoffee2,
                             model = ~Store + Temp + cost + type + size, trials = 54,
                             splitplotdesign = htcdesign, splitplotsizes = 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 sub-plots for gaussian models.

#Evaluate the design. Note the decreased power for the blocking factors.
## Not run: eval_design_mc(splitplotdesign, model = ~Store + Temp + cost + type + size, alpha = 0.05,
            blocking = TRUE, nsim = 100, glmfamily = "gaussian", varianceratios = c(1, 1))
## End(Not run)

#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")

## Not run: eval_design_mc(designbinom, ~a + b, alpha = 0.2, nsim = 100, effectsize = c(0.7, 0.9),
              glmfamily = "binomial")
## End(Not run)

#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:

## Not run: eval_design_mc(designpois, ~a + b, 0.05, nsim = 100, glmfamily = "poisson",
               anticoef = log(c(0.2, 2, 2)))
## End(Not run)


#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, alpha, nsim = 1000,
  distribution = "gaussian", censorpoint = NA, censortype = "right",
  rfunctionsurv = NULL, anticoef = NULL, effectsize = 2,
  contrasts = contr.sum, parallel = FALSE, detailedoutput = FALSE,
  advancedoptions = NULL, ...)
```

Arguments

`design`	The experimental design. Internally, all numeric columns will be rescaled to [-1, +1].
`model`	The statistical model used to fit the data.
`alpha`	The type-I error.
`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	If TRUE, uses all cores available to speed up computation of power. Default FALSE.
detailedoutput	If TRUE, return additional information about evaluation in results. Default FALSE.
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.
...	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 auto-generate 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(effec`
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,
  splitplotsizes = NULL, optimality = "D", augmentdesign = NULL,
  repeats = 20, varianceratio = 1, contrast = contr.simplex,
  aliaspower = 2, minDopt = 0.8, parallel = FALSE, timer = FALSE,
  splitcolumns = FALSE, randomized = TRUE, advancedoptions = 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 splitplotsizes, 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.

`splitplotsizes` Default NULL. Specifies the block size for each row of harder-to-change factors given in the argument `splitplotdesign`. If 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‘. If the input is a vector, each entry of the vector determines the size of the sub-plot for that whole plot setting. If the input is an integer, each block will be of that size.

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

`varianceratio` The ratio between the interblock and intra-block variance for a given stratum in a split plot design. Default 1.

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

`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.

`parallel` Default FALSE. If TRUE, the optimal design search will use all 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. 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.

`timer` Default FALSE. If TRUE, will print an estimate of the optimal design search time.

`splitcolumns` 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. If FALSE, the resulting design will be ordered from the left-most parameter.

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), and "progressBarUpdater" (a function called in non-parallel optimal searches that can be used to update an external progress bar).

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 attr function.

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)

#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.
## Not run:
options(cores = 2)
```

```
design2 = gen_design(categorical_candidates,
                     model = ~a + b + c, trials = 19, repeats = 1000, parallel = TRUE)

## End(Not run)

#You can also estimate the time it will take for a search to complete with by setting timer = TRUE.
## Not run:
design2 = gen_design(categorical_candidates,
                     model = ~a + b + c, trials = 500, repeats = 100, timer = TRUE)

## End(Not run)

#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 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 `splitplotsizes`. 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,
                          splitplotsizes = 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, splitplotsizes = 2,
                  varianceratio = 1, repeats = 10)

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

#Each Vineyard block has 4 runs with different Age:
## Not run:
splitsplitsplitplotdesign = gen_design(splitplotcandidateset2, ~Location + Climate + Vineyard + Age,
                                trials = 192, splitplotdesign = temp, splitplotsizes = 4,
                                       varianceratio = 1, splitcolumns = TRUE)

## End(Not run)
#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 designl, 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, splitplotsizes = 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 following attributes:

#D Efficiency
attr(design, "D")
#A Efficiency
attr(design, "A")
#The average prediction variance across the design space
attr(design, "I")
#G Efficiency
attr(design, "G")
#The minimum eigenvalue of the information matrix
attr(design, "E")
#The trace of the infomration matrix
attr(design, "T")
#The Alias Matrix
attr(design, "alias.matrix")
```

```
#The correlation matrix can be accessed via the "correlation.matrix" attribute:
correlation.matrix = attr(design2, "correlation.matrix")

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

## Not run: plot_correlations(design2)

#A fraction of design space plot can be produced by calling the plot_fds command

## Not run: 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)
```

`plot_correlations` *Plots design diagnostics*

Description

Plots design diagnostics

Usage

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

Arguments

`genoutput`	The output of either gen_design or eval_design/eval_design_mc
`model`	Default NULL. 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.

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 *Plots design diagnostics*

Description

Plots design diagnostics

Usage

```
plot_fds(genoutput, model = NULL, continuouslength = 11)
```

Arguments

genoutput	The run matrix
model	The model, by default uses the model used in eval_design or gen_design.
continuouslength	Default 9. The precision of the continuous variables.

Value

Plots design diagnostics

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)
```

---

skprGUI *Graphical User Interface for skpr*

Description

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

Usage

```
skprGUI(inputValue1, inputValue2)
```

Arguments

inputValue1	Required by Shiny
inputValue2	Required by Shiny

Examples

```
#Type skprGUI() to begin

## Not run: skprGUI()
```

---

skprGUIbrowser *skprGUIbrowser*

Description

skprGUI provides a graphical user interface to skpr, in an external browser.

Usage

```
skprGUIbrowser()
```

Examples

```
#Type skprGUIbrowser() to begin

## Not run: skprGUIbrowser()
```

skprGUIserver *Graphical User Interface for skpr*

Description

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

Usage

```
skprGUIserver(inputValue1, inputValue2)
```

Arguments

inputValue1	Required by Shiny
inputValue2	Required by Shiny

Examples

```
#Type skprGUIserver() to begin

## Not run: skprGUIserver()
```

%>% *re-export magrittr pipe operator*

Description

re-export magrittr pipe operator

Index