Journal of Statistical Software

August 2021, Volume 99, Issue 1. doi: 10.18637/jss.v099.i01

Optimal Design Generation and Power Evaluation in R: The skpr Package

Tyler Morgan-Wall
Institute for Defense Analyses

George Khoury
Institute for Defense Analyses

Abstract

The R package **skpr** provides a suite of functions to generate and evaluate experimental designs. Package **skpr** generates D, I, Alias, A, E, T, and G-optimal designs, and supports custom user-defined optimality criteria, N-level split-plot designs, mixture designs, and design augmentation. Also included are a collection of analytic and Monte Carlo power evaluation functions for normal, non-normal, random effects, and survival models, as well as tools to plot fraction of design space plots and correlation maps. Additionally, **skpr** includes a flexible framework for the user to perform custom power analyses with external libraries and user-defined functions, as well as a graphical user interface that wraps most of the functionality of the package in a point-and-click web application.

Keywords: design of experiments, optimal design, power, R.

1. Introduction

The science of design of experiments (DOE) includes many types of experimental designs, including factorial, fractional factorial, screening, and optimal designs. Optimal designs maximize, for a given number of test runs, an objective criterion of the model to be estimated. Whereas some design types constrain users to certain set numbers of runs for a given set of factors, the optimal design approach to test planning allows for search for the "best" design for any user-specified number of runs.

A design can be optimal for a given design criterion and specified number of runs but still be inadequate for the actual experimental goal. The goal in running an experiment is to obtain insight into how the response variable is affected by changes in the experimental factors, and an experimenter should evaluate their designs to determine if the planned sample size is large enough to detect these effects. A useful measure of design quality is the statistical *power* of the design. Power is defined as $1 - \beta$, where β is the type-II error rate (false negative rate),

which is a function of the model matrix and the statistical hypotheses (null and alternate) being tested. By maximizing power, the experiment is more likely to detect an effect if one actually exists. These power values can be calculated analytically for linear models, and calculated via Monte Carlo simulation for generalized linear models and survival models.

Package **skpr** (Morgan-Wall and Khoury 2021) is an open-source R package available from the Comprehensive R Archive Network at `https://CRAN.R-project.org/package=skpr` that provides both optimal design generation and power evaluation tools. It improves upon existing algorithmic design generation packages in R by providing a wider variety of design generation options and improved search algorithms, power evaluation facilities, plotting options, and an optional graphical user interface. In particular, **skpr** provides functions to perform the following tasks:

1. Generate D, I, Alias, A, E, T, and G-optimal designs, as well as user-defined optimality criteria.
2. Generate optimal split/split-split/N-level split-plot designs with no limit to the number of split-plot strata.
3. Evaluate the statistical power of these designs, assuming a normal response.
4. Evaluate, with Monte Carlo simulation, the power of designs with non-normal response variables, survival analyses, and user-provided fitting libraries.
5. Evaluate the power of split-plot designs with normal and non-normal response variables with no limit to the number of sub-plot levels.

There are many R packages for design generation, but few for optimal design generation. **AlgDesign** (Wheeler 2019) is the only R package that directly offers optimal design generation in an open-source package: other packages that offer this functionality implement their routines by calling **AlgDesign** (e.g., **DoE.base**, Grömping 2018) or by calling external, closed-source libraries to perform the optimization, as package **OptimalDesign** (Harman and Filova 2019) does with the commercial **Gurobi** optimization software (**Gurobi** Optimization, LLC 2019). Package **DoE.base** also offers tools to generate factorial designs with or without blocking and orthogonal arrays for main effects experiments. For more traditional DOE non-factorial designs when users have constrained resources and cannot run a full factorial design, **FrF2** (Grömping 2014b), **FrF2.catlg128** (Grömping 2013), **conf.design** (Venables 2013), and **planor** (Kobilinsky, Bouvier, and Monod 2020) offer tools for generating fractional factorial designs. For response surface designs, package **rsm** (Lenth 2009) provides tools to generate and evaluate designs for response surface models.

For design evaluation, there are many R packages that offer a variety of evaluation functions, but few that offer power analysis tools for generic experimental designs. The **pwr** package (Champely 2020) offers tools to calculate power for statistical tests, but no functions for full experimental designs. Package **simr** (Green and MacLeod 2016) provides functions to perform Monte Carlo power simulations for generalized linear models focusing on pilot studies and fitted models. Outside of R, the standalone power analysis tool **G*Power** (Faul, Erdfelder, Lang, and Buchner 2007) can calculate power for a wide variety of tests and effect sizes and generate plots of power curves versus effect size and sample size, but it provides no Monte Carlo support. Commercially, **JMP** (SAS Institute Inc. 2018) and **Design-Expert** (Stat-Ease,

Inc. 2021) both provide excellent design generation and power evaluation tools, but limited or no built-in support for Monte Carlo power simulation.

Package **skpr** is primarily made up of five user-accessible functions:

1. `gen_design` – Optimal design generation.
2. `eval_design` – Analytic (normal) power evaluation.
3. `eval_design_mc` – Generalized linear model Monte Carlo power.
4. `eval_design_survival_mc` – Survival Monte Carlo power.
5. `eval_design_custom_mc` – Custom library Monte Carlo power.

Additionally, **skpr** provides two plotting functions for evaluating designs:

6. `plot_fds` – Fraction of design space plot.
7. `plot_correlation` – Correlation map plot.

Also provided is a graphical user interface (GUI), `skprGUI()`, that provides a web interface through the **shiny** package (Chang *et al.* 2021) for a subset of the above functionality. The only existing DOE R package that offers a GUI is **RcmdrPlugin.DoE** (Grömping 2014a), a plugin for the **RCmdr** (Fox 2005) GUI packages that provides access to a selection of the DOE analysis tools offered in the **DoE.base**, **FrF2**, and **DoE.wrapper** (Grömping 2020) packages. In particular, this package provides optimal design generation based on **AlgDesign**'s D-optimal design search algorithm. There are no other packages in the R ecosystem that offer a similar GUI to access the design generation and power evaluation features in **skpr**.

The typical workflow for the package is to generate an optimal design with `gen_design()` and then evaluate it with the one of the evaluation functions. However, the user can also import designs generated elsewhere and evaluate them within **skpr**; evaluating designs from external sources is no different than evaluating designs produced by **skpr**, as a design is simply a `data.frame` with the proper variable types for each column and, if applicable, a blocking structure encoded somewhere in the data structure. The output of the evaluation functions are tidy data frames (Wickham 2014) with the parameters listed with their powers and the calculation type (e.g., `parameter.power.mc` indicates a parameter power is calculated via Monte Carlo).

2. Generating designs

The function `gen_design()` generates optimal designs for a given set of factors, model, number of runs, and optimality criterion. It generates optimal designs using a point exchange algorithm, which takes as input a data frame of the permissible test points, called the candidate set, and from that generates a design that optimizes the chosen optimality criterion (giving a design that is "best" in some sense). The criteria involve maximizing (or minimizing) functions of the model matrix X and the moments matrix M (Studden 1977):

$$M = \int_{x \in \chi} f(x) f'(x)\, dx,$$

where $\chi = [-1, +1]^k$ is the design region for k factors, and $f(x)$ is a function that takes a vector of factor settings and expands it to its corresponding model terms.

Package **skpr** supports eight design criteria:

1. D-optimal designs minimize $|(X^\top X)^{-1}|$, or equivalently maximize $|X^\top X|$, the determinant of the information matrix (Atkinson and Donev 1992).

2. I-optimal designs minimize the average prediction variance over the design space, $\text{tr}[(X^\top X)^{-1} M]$.

3. Alias-optimal designs minimize the trace of the sum of squares of the alias matrix A, $\text{tr}(A^\top A)$ (where $A = (X^\top X)^{-1} X^\top X_2$ and X_2 is the model matrix representing the interaction columns), while simultaneously ensuring the D-optimality does not drop below a certain user-defined threshold. This function can generate designs with a favorable aliasing structure, which enables the user to conduct screening experiments for active effects (Jones and Nachtsheim 2011b).

4. A-optimal designs minimize $\text{tr}[(X^\top X)^{-1}]$, the trace of the inverse of the information matrix. This criterion results in minimizing the average variance of the estimates of the regression coefficients (Atkinson and Donev 1992).

5. G-optimal designs minimize the maximum entry in the diagonal of $X(X^\top X)^{-1} X^\top$, the hat matrix. This minimizes the maximum variance of the predicted values (Atkinson and Donev 1992).

6. E-optimal designs maximize the minimal eigenvalue of the information matrix $X^\top X$, which minimizes the worst-case variance of any linear combination of estimated coefficients (Atkinson and Donev 1992).

7. T-optimal designs maximize the trace of the information matrix, $\text{tr}(X^\top X)$ (Atkinson and Donev 1992).

8. Custom-optimal designs maximize a function of the design's model matrix defined by the user, $f(X)$.

The design found by `gen_design()` will depend on the candidate set provided. For an unconstrained design, the user should provide a full factorial design for the test factors; for a design with constraints, the user still generates a full factorial candidate set, but then removes the design points that do not satisfy the constraints. Generating a candidate set in R is most easily performed by calling the R base function `expand.grid()` with a list of factors and their corresponding levels. This generates a data frame of all combinations of the input factors. Continuous factors are type `numeric` and the user must specify which points in the interval are included; categorical factors are either type `character` or `factor`.

To start the search process, `gen_design()` first generates an initial design by randomly sampling from the candidate set. If there are fewer experimental runs than points in the candidate set, **skpr** does this initial sampling without replacement; if there are more experimental runs than points in the candidate set, the sampling is done with replacement. If the initial design created is singular, this random generating process is repeated $10 \times$ `trials` times. If that

process also fails to generate an initial non-singular design, **skpr** switches to a nullification algorithm to generate non-singular design. This algorithm (Wheeler 2019) finds a non-singular design (if one exists) from the candidate set using the Gram-Schmidt orthogonalization procedure. Function `gen_design()` then runs modified Federov's search algorithm (Cook and Nachtrheim 1980) and exchanges rows from the design with rows in the candidate set (with replacement) until it can no longer improve the optimality criterion more than a user-adjustable minimal level (default 10^{-5}). To increase the probability that it has found a globally optimal design `gen_design()` records the design and repeats the entire search `repeats` times (default 20, but can be specified by the user), randomizing the initial design each time. The algorithm then returns the best design according to the optimality criterion chosen.

The following R code gives a simple example of using **skpr** to generate a D-optimal design. For reproducibility, it is important to set the random seed. The example also shows the use of `get_attribute()` and `get_efficiency()` to extract the information **skpr** stores in the attributes of the design:

```
R> set.seed(1131993)
R> candidateset <- expand.grid(X1 = c(-1, 0, 1), X2 = c(-1, 0, 1),
+    X3 = c(-1, 0, 1))
R> design <- gen_design(candidateset = candidateset,
+    model = ~ X1 + X2 + X3, trials = 8)
R> get_attribute(design, "variance.matrix")

     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    0    0    0    0    0    0    0
[2,]    0    1    0    0    0    0    0    0
[3,]    0    0    1    0    0    0    0    0
[4,]    0    0    0    1    0    0    0    0
[5,]    0    0    0    0    1    0    0    0
[6,]    0    0    0    0    0    1    0    0
[7,]    0    0    0    0    0    0    1    0
[8,]    0    0    0    0    0    0    0    1

R> get_optimality(design)

    D    I   A            G  T E Alias
1 100 0.25 100 Not Computed 32 8     3
```

If none of the built-in optimality criteria suffice, `gen_design()` also allows the user to specify their own optimality criterion with the `custom` optimality option. With this option the user is able to specify their own function of the model matrix $f(X) =$ `customOpt(X)` in R, which is used when optimizing the design. This is slower than any of the built-in options because the underlying C++ code (using **Rcpp** and **RcppEigen**, Eddelbuettel 2013; Bates and Eddelbuettel 2013) has to call back to the R environment, but it provides additional flexibility to the user.

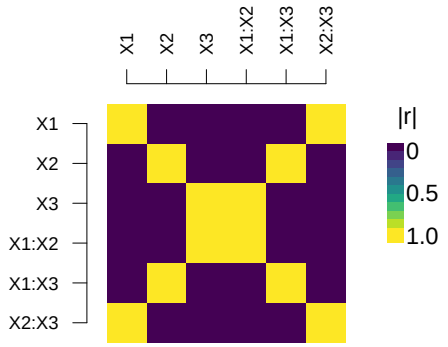
There are a few optimality-dependent changes to the search process: for G-optimal designs, **skpr** first finds a D-optimal design and then uses that as the initial design for the G-optimal search. This is because D and G-optimal designs are highly correlated across the exact

X1	X2	X3
−1	−1	−1
−1	−1	−1
−1	1	1
−1	1	1
1	−1	1
1	−1	1
1	1	−1
1	1	−1

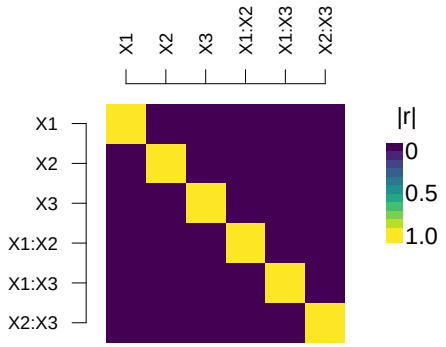
(a)

X1	X2	X3
−1	−1	−1
−1	−1	1
−1	1	−1
−1	1	1
1	−1	−1
1	−1	1
1	1	−1
1	1	1

(b)



(c)



(d)

Figure 1: (a) 100% D-efficient design with $\text{tr}(A^\top A) = 6$. (b) 100% D-efficient design with $\text{tr}(A^\top A) = 3$. This is a full-factorial design, and is the classical orthogonal design for this combination of factors. (c) Correlation map for the design in part (a): While there is no correlation between the main effects, there is perfect correlation in the interaction terms. This did not degrade the design's D-optimality, which only depends on the terms included in the model when generating the design. (d) Correlation map for the design in part (b): This design is superior due to having both an orthogonal correlation structure in its main effect terms as well as the interaction terms. This difference is not captured in the optimality, but is captured via the alias matrix.

design space, although not equivalent as they are when using approximate design theory (Atkinson and Donev 1992). For T, G, and E-optimal designs, an additional singularity check is performed at each exchange: if the design is singular, the exchange is not performed.

If there is a tie among the best designs found (e.g., two designs each with the same D-optimality), `gen_design()` calculates the trace of the sum of squares of the alias matrix and uses that value as a tie breaker. A lower value corresponds to less bias from terms not included in the model but that are active in the experiment. This tie-breaker is used for everything but Alias-optimal designs (where this metric is already being directly minimized). As an example, both of the designs in Figure 1 are 100% D-efficient, but the second has a better aliasing structure. To speed up computation, `gen_design()` supports multiple cores by setting the argument `parallel = TRUE`. This seamless multicore capability is implemented using the **parallel** (R Core Team 2021) and **foreach** (Microsoft and Weston 2020; Kane, Emer-

son, and Weston 2013) packages, as well as the **doRNG** (Gaujoux 2020) package to ensure reproducibility when calculating on multiple cores with a user-defined seed. The user can either specify the number of cores manually by setting `options(cores = #)`, or use the default (which uses all the cores available). This can reduce the running time of the search up to a factor of N_{cores}.

2.1. Mixture designs

A mixture experiment is an experiment where the important factors are the relative proportions of the components and not the total quantity (Scheffé 1958). For example, this can be as simple as a recipe with different proportions of ingredients, or an industrial experiment where the response is the tensile strength of steel with different mixtures of iron, nickel, copper, and chromium. The general purpose of mixture experimentation, in combination with response surface methodology, is for practitioners to estimate what proportion of inputs from the entire design space works "best" in some sense, with only a limited number of experimental runs (Cornell 1973).

In **skpr**, searching for a mixture design is no different than searching for any other type of constrained design, as the underlying point exchange algorithm is unchanged. The only difference is the model must not include the intercept, as this term is perfectly correlated with the sum of the component parts of the mixture, and each row of the candidate set must satisfy the mixture constraint. In R, removing the intercept is done by adding `-1` to the model formula (e.g., `~. + -1`). The mixture constraint where all components in each potential run need to add to one is introduced as any other constraint is: by filtering the candidate set prior to passing it into `gen_design()` so that each entry in the candidate set fulfills the constraint.

As an example, suppose a user wants to experiment on a cookie recipe by varying the proportions of each ingredient. The initial candidate set features all possible combinations of the various ingredient proportions, ignoring the mixture constraint that all of the components must sum to one. The user would then filter the candidate set using the constraint, and then pass the filtered candidate set to `gen_design()` to generate a mixture design. Here's an example of how a user might generate and filter a candidate set, and then generate an experimental design with `gen_design()`:

```
R> candset <- expand.grid(flour = seq(0.2, 0.8, 0.01),
+    sugar = seq(0.5, 0.8, 0.01), bakingsoda = seq(0.2, 0.8, 0.01))
R> mixtureset <- candset[with(candset, flour + sugar + bakingsoda == 1), ]
R> gen_design(mixtureset, ~ -1 + flour + sugar + bakingsoda, 8)

  flour sugar bakingsoda
1   0.2   0.6        0.2
2   0.3   0.5        0.2
3   0.2   0.5        0.3
4   0.2   0.6        0.2
5   0.3   0.5        0.2
6   0.2   0.5        0.3
7   0.2   0.6        0.2
8   0.2   0.5        0.3
```

2.2. Blocked and split-plot designs

A common constraint on an experimental design is that one of the factors is hard-to-change, and thus must be varied less often than the other factors (Ju and Lucas 2002). In such a design, this "hard-to-change" factor is kept constant for some number of runs while the "easy-to-change" factors vary from run to run (Yates 1935; Jones and Nachtsheim 2009; Goos and Vandebroek 2001; 2003; 2005). This is called a **split-plot** design, and a set of runs in which the hard-to-change factor is held constant is called a whole-plot. These designs were originally developed by Fisher (1925) for use in agricultural experiments (the "plot" in split-plot is in reference to the plots of land into which these experiments were divided). A split-plot design significantly changes the noise structure of the model, and consequently changes the power of factors within the design as well. In certain cases, these types of designs can be more efficient than the corresponding completely randomized design, both in terms of experimental cost and statistical inference (Goos and Vandebroek 2004).

Additionally, there may also be additional nuisance parameters that increase variability in the response, but are not of primary interest to the experimenter. When runs that share this nuisance factor are grouped together, it can be worthwhile make the grouping explicit. These groups are called blocks, and accounting for them during design generation (when it is known that they will be present during the experiment) can lead to a more efficient design (Goos and Jones 2011).

For these designs, the parameter estimate is no longer the ordinary least squares (OLS) estimator,

$$\hat{\beta} = (X^\top X)^{-1} X^\top Y,$$

but rather the generalized least squares (GLS) estimate (also named the feasible GLS estimator) (Aitken 1936):

$$\hat{\beta} = (X^\top V^{-1} X)^{-1} X^\top V^{-1} Y,$$

where V is the covariance matrix of the response vector, which **skpr** calculates automatically from the blocking structure of a design. The D-optimal condition is no longer to maximize the determinant of the information matrix $X^\top X$, but rather the determinant of $X^\top V^{-1} X$. This extends to other optimal conditions; the insertion of the V^{-1} in the optimality criteria is as follows in **skpr**:

1. D-optimal: maximize $|X^\top V^{-1} X|$.
2. I-optimal: minimize $\text{tr}[(X^\top V^{-1} X)^{-1} M]$.
3. A-optimal: minimize $\text{tr}[(X^\top V^{-1} X)^{-1}]$.
4. G-optimal: minimize the maximum diagonal element of $X(X^\top V^{-1} X)^{-1} X^\top V^{-1}$.
5. E-optimal: maximize the minimal eigenvalue of $X^\top V^{-1} X$.
6. T-optimal: maximize $\text{tr}(X^\top V^{-1} X)$.
7. Custom-optimal: maximize a user-defined function $f(X, V^{-1})$.

For other discussions of D and I split-plot optimality criteria, see Letsinger, Myers, and Lentner (1996). The custom-optimal option leaves the specification of the optimality up to

the user (as a function of X, the model matrix). For split-plot designs, the user can specify a function of both X and V^{-1} as their optimality criterion.

V is both a function of the ratio ρ of the block-to-block variance σ_b^2 and the run-to-run variance σ^2 (Letsinger *et al.* 1996). The block-to-block variance is often substantially larger than the run-to-run variance, but common practice is to set the ratio ρ to one (Goos 2002). This is justified by empirical observations that the D-optimal design is not particularly sensitive to the specific value used (Goos and Vandebroek 2001), and that a design that is D-optimal for one ratio is D-optimal for a wide range of ratios, as long as the value of ρ is not too close to zero (Goos and Vandebroek 2003).

Package **skpr** builds optimal split-plot designs using a step-wise method, starting by generating optimal designs for the factors that are hardest-to-change and ending with the factors that are easiest-to-change. For the hardest-to-change factors, the algorithm generates an optimal design for just those factors, ignoring all of the sub-plots. This is because the whole-plots terms usually have fewer degrees of freedom than the easy-to-change terms, and thus generally have less statistical power. Thus, any exchange sacrificing the optimality of a whole-plot for a sub-plot is likely a poor one. The sub-plots are generated with the harder-to-change factor settings fixed, and thus the algorithm searches for the most optimal sub-plot factor level settings given that constraint.

To generate an optimal split-plot design with `gen_design()`, the user needs to specify one additional input:

1. The hard-to-change design (the argument `splitplotdesign`). This is a data frame consisting of the design generated for the hard-to-change factors. It can either be an ordinary design or one with existing sub-plots, as long as the nesting structure is stored in the row names. One row of this design specifies the hard-to-change factor settings for an entire whole plot in the overall design.

Also, optionally, the user can specify two additional arguments:

2. A number specifying the number of runs in each plot (the argument `blocksizes`). By default, this is calculated automatically given the number of runs desired in the full design (`trials`) and the number of rows in `splitplotdesign`. `gen_design()` creates a design that is as balanced as possible. If the user wants to manually specify the size of each block, they can pass in a vector of length `nrow(splitplotdesign)` that specifies a size for each sub-plot. This vector must sum up to the total number of trials specified in `trials`.

3. A scalar (the argument `varianceratios`) specifying the ratio of the block-to-block variance for the blocks specified in `splitplotdesign` to the run-to-run variance. By default, this is 1.

The variance-covariance matrix V is constructed at each step of the split-plot design creation process, using the values provided by the user (or by default) in argument `varianceratios`. As an example, for a 12-run design with four whole-plots and three runs per whole-plot:

```
R> candset <- expand.grid(X1 = c(1, -1), X2 = c(1, -1), X3 = c(1, -1))
R> htc_design <- gen_design(candset, ~ X1, 4)
```

$$V = \begin{pmatrix}
5 & 4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 5 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 4 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 5 & 4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 4 & 5 & 4 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 4 & 4 & 5 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 5 & 4 & 4 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 4 & 5 & 4 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 4 & 4 & 5 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5 & 4 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 5 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 4 & 5
\end{pmatrix}$$

Figure 2: The variance-covariance matrix for a 12-run split-plot design with four whole-plots and three sub-plots in each whole-plot. The ratio of block-to-block and run-to-run variance in this case is four.

```
R> fulldesign <- gen_design(candset, ~ X1 + X2 + X3, 12,
+    splitplotdesign = htc_design, varianceratio = 4)
R> fulldesign
```

```
    X1 X2 X3
1.1  1  1 -1
1.2  1  1  1
1.3  1 -1  1
2.1 -1  1 -1
2.2 -1  1  1
2.3 -1 -1 -1
3.1  1  1 -1
3.2  1 -1 -1
3.3  1 -1  1
4.1 -1  1  1
4.2 -1 -1  1
4.3 -1 -1 -1
```

and the variance-covariance matrix is shown in Figure 2. The matrix in Figure 2 shows that there is correlation between runs within a whole-plot, but not between whole-plots. For a split-split-plot design, this process is repeated for the next layer, with this design as an input. If this design is instead divided into three whole-plots, each with two sub-plots that each have two sub-plots of their own (and each with a variance ratio of one), the resulting design is as follows (note the additional layer separated by a period in the row names):

```
R> candset <- expand.grid(X1 = c(1, -1), X2 = c(1, -1), X3 = c(1, -1))
R> vhtc_design <- gen_design(candset, ~ X1, 3)
R> htc_design <- gen_design(candset, ~ X1 + X2, 6,
+    splitplotdesign = vhtc_design, varianceratio = 4)
```

$$V = \begin{pmatrix}
7 & 6 & 4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 7 & 4 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 4 & 7 & 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 4 & 6 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 7 & 6 & 4 & 4 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 6 & 7 & 4 & 4 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 4 & 4 & 7 & 6 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 4 & 4 & 6 & 7 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 6 & 4 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & 7 & 4 & 4 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 4 & 7 & 6 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 4 & 6 & 7
\end{pmatrix}$$

Figure 3: The variance-covariance matrix for a 12-run split-split-plot design with three whole-plots and two sub-plots in each whole-plot, and two additional sub-sub-plots in each sub-plot. The block-to-block variance ratios (relative to the run-to-run variance) in this case are four for the whole plots and two for the sub-plots.

```
R> fulldesign <- gen_design(candset, ~ X1 + X2 + X3, 12,
+    splitplotdesign = htc_design, varianceratio = 2)
R> fulldesign
```

```
      X1 X2 X3
1.1.1 -1 -1  1
1.1.2 -1 -1 -1
1.2.1 -1  1 -1
1.2.2 -1  1  1
2.1.1  1  1 -1
2.1.2  1  1  1
2.2.1  1 -1  1
2.2.2  1 -1 -1
3.1.1  1 -1 -1
3.1.2  1 -1  1
3.2.1  1  1  1
3.2.2  1  1 -1
```

and the resulting variance-covariance matrix V is shown in Figure 3. In the generated designs, the blocking information is encoded in the row name structure with periods: from left to right, the row number indicates the whole-plot, sub-plot, sub-sub-plot, etc. Rather than designate a special column for blocking information, storing the blocking structure in the row names keeps it visible to the user but separate from the actual factor levels in the design. However, if the user wants to specify their blocking information that way, **skpr** supports conversion of columns titled `Block#`, where `#` indicates the blocking level in the design, as well as columns labeled `Whole Plot` and `Sub Plot`, for compatibility with designs generated in **JMP**. Package **skpr** converts these columns into the row name structure used internally, and removes them

from the design to maintain the separation of blocking structure and the actual factor level settings.

For split-plot designs with interactions between whole-plots and sub-plots, `gen_design()` computes the interaction term of the model matrix programmatically during the search process. `gen_design()` parses the model, determines if there are interactions between whole-plots and sub-plots, and then passes those interaction terms to the point exchange algorithm. Those columns in the model matrix are then computed on-the-fly during the search process. `gen_design()` supports disallowed combinations both within a sub-plot and between whole-plots and sub-plots.

2.3. Design augmentation

Sequential experimental design is an important tool when designing resource-constrained experiments. Sequential design refers to running a much smaller (and thus, less resource-intensive) screening design to test for the existence of effects before committing to a more expensive experiment to characterize their size. After running a screening experiment and detecting that some effects are active, a practitioner needs to design a follow-up experiment to actually characterize the active effects. Generating an entire new optimal design ignores the information already collected in the prior experiment, wasting testing resources that can be targeted at the effects of interest. `gen_design()` supports design augmentation (Nachtsheim and Jones 2018), which allows the user to set a pre-existing design and generate additional optimal runs, given the fixed design. The user can simply pass the pre-existing design into `gen_design()` with the argument `augmentdesign`, and `gen_design()` sets those runs in place and attempts to find the best $N_{\text{total}} - N_{\text{augment}}$ runs to add to the design.

As an example, suppose 12 experimental runs have already been executed given a screening design with six continuous factors. The original screening experiment was performed and the user wants to generate additional runs to investigate the entire model. The initial Alias-optimal design is (reordering the rows to display the design's structure):

```
R> set.seed(1)
R> candidateset <- expand.grid(X1 = c(1, 0, -1), X2 = c(1, 0, -1),
+    X3 = c(1, 0, -1), X4 = c(1, 0, -1),
+    X5 = c(1, 0, -1), X6 = c(1, 0, -1))
R> screening_design <- gen_design(candidateset, ~., 12, repeats = 100,
+    optimality = "Alias")
R> pair_zeros <- function(design) {
+    as.vector(apply(design, 2, (function(x) order(abs(x))))[1:2, ])
+  }
R> screening_design_ordered <- screening_design[pair_zeros(screening_design),]
R> screening_design_ordered

   X1 X2 X3 X4 X5 X6
9   0  1 -1 -1 -1  1
11  0 -1  1  1  1 -1
1   1  0  1 -1  1  1
8  -1  0 -1  1 -1 -1
4   1 -1  0  1 -1  1
```

```
6  -1  1  0 -1  1 -1
5  -1 -1 -1  0  1  1
7   1  1  1  0 -1 -1
10  1 -1 -1 -1  0 -1
12 -1  1  1  1  0  1
2  -1 -1  1 -1 -1  0
3   1  1 -1  1  1  0
```

Here, `gen_design()` algorithmically generated a design known as a *definitive screening design* (Jones and Nachtsheim 2011a). To augment this design, we input this design as the argument `augmentdesign` in `gen_design()` and generate a new D-optimal design with additional runs and a full model with all interactions. Augmenting the design also includes a blocking column, separating the original sets of runs into one block and the new set of runs into a second block. We can compare this augmented design with one generated from scratch by comparing their D-efficiencies (adding a blocking column manually to the hand-built design to account for the added block structure):

```
R> aug_design <- gen_design(candidateset, ~.*., 34,
+    augmentdesign = screening_design_ordered, optimality = "D")
R> no_aug_design <- gen_design(candidateset, ~.*., 22, optimality = "D")
R> aug_design %>%
+    eval_design(model = ~.*., alpha = 0.2, blocking = TRUE) %>%
+    attr("D")

[1] 74.49612

R> rbind(screening_design_ordered, no_aug_design) %>%
+    cbind(data.frame(Block1 = c(rep(1,12),rep(2,22)))) %>%
+    eval_design(model = ~.*., alpha = 0.2, blocking = TRUE) %>%
+    attr("D")

[1] 71.66363
```

The augmented design has a higher D-efficiency than the design generated without taking the original screening experiment into consideration. Additionally, design augmentation allows users more flexibility in choosing a follow-up experiment: for the model with all two-factor interactions, generating a design from scratch requires the new design to have at least 22 runs to be non-singular, for a total of 34 runs. For augmenting the original 12-run design with the new model, the follow-up experiment does not need to be non-singular on its own: only the design as a whole needs to be non-singular. Thus, the user has the flexibility to add as few as 10 additional runs and still be able to fit the new model, fewer than half of what is required when generating a standalone follow-up design.

2.4. Comparison with AlgDesign

Package **AlgDesign**, although not under active development, is the most prominent R package (as of 2021) that natively offers flexible optimal design generation and does not depend

on commercial software. Like **skpr**, **AlgDesign** uses an exchange algorithm to search for optimal designs, offering the ability to produce D, I, and A-optimal designs (with function `AlgDesign::optFederov()`), as well as the ability to construct split-plot designs sequentially. **AlgDesign** also provides approximate design generation, which is not included in **skpr**.

A key difference in the algorithms used by each package is the method of selecting points from the candidate set. **skpr** searches for designs from the candidate set with replacement, while **AlgDesign** searches without replacement. For a certain subset of optimal designs (D-optimal, number of trials fewer than the full factorial design for the model) **AlgDesign**'s approach is reasonable, as it is unlikely that a D-optimal design for a linear model has repeated design points. However, there is a major problem with this algorithm as applied to many optimal design searches: some designs require repeated points. Examples include I-optimal response surface designs, which can have repeated center points, or designs with more trials than points in the candidate set. A user can work around this issue by duplicating points in the candidate set, but this requires the user to anticipate the need for duplication. For large candidate sets, this workaround can also make the design search prohibitively slow.

3. Evaluating power for designs

Design generation is only the first step; once a design is created, the user must evaluate it to determine if it is adequate for the hypothesis being tested. An optimal design is optimal in the sense that it maximizes/minimizes a chosen function of the information matrix, but that does not mean it is adequate for the experiment. This is because the quality of the experiment depends on factors outside of the design: namely, the effect size and variability of the response variable. We thus want to evaluate a design's ability to detect an effect given that one is present, otherwise known as *statistical power*. Maximizing power is the same as minimizing type-II error, the rate of false negatives.

Power analysis starts with the linear model used to fit the experiment:

$$Y = X\beta + \epsilon,$$

where Y is the vector of responses, X is the model matrix, β is the model parameter vector, and ϵ is the run-to-run variability. **skpr** assumes that the variance of ϵ is one for all its calculations, and the discussion that follows also sets $\sigma_\epsilon^2 = 1$. We will test the hypothesis that $\beta_i = 0$ versus the alternative $\beta_i \neq 0$, where β is the coefficient vector that contains one or more coefficients for each factor or interaction effect, and i indicates the subset of coefficients corresponding to the ith factor. Here, X has size $N \times p$ where N is the number of runs and p is the number of parameters in the model, and β has size $p \times 1$. The test statistic is

$$F_0 = \frac{\hat{\beta}_i^{\,2}}{\hat{\sigma}^2 (X^\top X)^{-1}_{ii}}.$$

This test statistic has an F-distribution with (g_i, $N - p - 1$) degrees of freedom, where g_i is the number of levels in the factor minus one (for continuous factors, g_i is 1). For the null hypothesis, the anticipated coefficients are zero and the test statistic follows a central F-distribution; in this case, we reject the null when $F_0 > \Delta_c = F^{-1}(1 - \alpha, g_i, N - p)$. Under the alternative hypothesis the test statistic follows a non-central F-distribution, $F_0 \sim$

$F(g_i, N-p, \lambda_i)$, and the power of the test is:

$$P(F_0 > \Delta_c) = 1 - F(F^{-1}(1-\alpha, g_i, N-p), g_i, N-p, \lambda_i),$$

where α is the allowable type-I error rate, and λ_i is the non-centrality parameter for the ith factor. The non-centrality parameter is equal to the ratio of the between group variance to the within group variance. The within group variance for the ith factor is defined as the sub-matrix of $(X^\top X)^{-1}$ that corresponds to the levels of the ith factor. By isolating this sub-matrix with a *hypothesis matrix* L_i and similarly isolating the components of the anticipated coefficients corresponding to the ith factor, we can calculate this ratio.

The hypothesis matrix takes the form $L_i = [A\ B\ C]$. A is a matrix of zeros of size $g_i \times E$, where E is the number of parameters in the model preceding the i-th effect. B is the identity matrix of size $g_i \times g_i$, and C is a matrix of zeros of size $g_i \times F$, where F is the number of parameters in the model following the i-th effect. In effect, L_i extracts just the parameters of interest from the coefficient vector or information matrix. As an example, suppose that $\beta = [1\ \beta_{11}\ \beta_{12}\ \beta_{21}\ \beta_{22}\ \beta_{23}]^\top$, and we wish to construct L_1 to extract just the β_{11} and β_{12} coefficients, then the hypothesis matrix would be

$$L_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix},$$

such that $L_1\beta = [\beta_{11}\ \beta_{12}]^\top$.

In matrix form, the non-centrality parameter for the ith factor is

$$\lambda_i = (L_i\beta)^\top (L_i(X^\top X)^{-1}L_i^\top)^{-1} L_i\beta.$$

For parameter estimates, we replace the $g_i \times p$ hypothesis matrix with a $1 \times p$ matrix, which extracts one column corresponding to the parameter of interest from the model matrix.

The following code provides a simple example of design generation and evaluation with **skpr**. By default, `eval_design()` assumes that the alternative hypothesis is $\beta_i = 1$ for all i, but this can be changed with the `anticoef` or `effectsize` arguments. Unless otherwise specified, `eval_design()` defaults to the model used during design generation and `alpha = 0.05`. Information about the evaluation is printed below the results.

```
R> candidate_set <- expand.grid(temp = c(80, 85, 90),
+    roast = c("Light", "Medium", "Dark"), brewtime = c(60, 120, 180))
R> design <- gen_design(candidate_set,
+    model = ~ temp + roast + brewtime + I(brewtime ^ 2), trials = 12)
R> eval_design(design)

       parameter            type     power
1    (Intercept)    effect.power 0.3774783
2           temp    effect.power 0.8212779
3          roast    effect.power 0.4605264
4       brewtime    effect.power 0.6295236
5  I(brewtime^2)    effect.power 0.2665443
6    (Intercept) parameter.power 0.3774783
```

```
7           temp parameter.power 0.8212779
8         roast1 parameter.power 0.5118998
9         roast2 parameter.power 0.5118998
10      brewtime parameter.power 0.6295236
11 I(brewtime^2) parameter.power 0.2665443
=============Evaluation Info==============
• Alpha = 0.05 • Trials = 12 • Blocked = FALSE
• Evaluating Model = ~temp + roast + brewtime + I(brewtime ^ 2)
• Anticipated Coefficients = c(1, 1, 1, -1, 1, 1)
```

We can also do this over a range of inputs for both effect size and sample size to plot power as a function of the various design parameters. In this way `eval_design()` can be used to determine what is an adequate sample size for a user's experiment, by repeatedly calculating power until a design is obtained in which all of the parameters of interest fall below the allowable type-II error rate. For example, if the user wanted to ensure that all parameters in their model exceeded 80% power, they can iteratively generate optimal designs and evaluate power in a `for` loop. Here, we take advantage of the `detailedoutput = TRUE` argument in `eval_design()`, which includes additional information in the output that can be useful when plotting (example shown in Figure 4).

```
R> counter <- 1
R> designpower <- list()
R> for(samplesize in 12:40) {
+    design <- gen_design(candidate_set,
+      model = ~ temp + roast + brewtime + I(brewtime ^ 2),
+      trials = samplesize)
+    designpower[[counter]] <- eval_design(design, detailedoutput = TRUE)
+    counter <- counter + 1
+  }
R> designpower[[length(designpower)]]

       parameter            type     power anticoef alpha trials
1    (Intercept)    effect.power 0.9522632       NA  0.05     40
2           temp    effect.power 0.9999853       NA  0.05     40
3          roast    effect.power 0.9946866       NA  0.05     40
4       brewtime    effect.power 0.9985188       NA  0.05     40
5  I(brewtime^2)    effect.power 0.8331067       NA  0.05     40
6    (Intercept) parameter.power 0.9522632        1  0.05     40
7           temp parameter.power 0.9999853        1  0.05     40
8         roast1 parameter.power 0.9879933        1  0.05     40
9         roast2 parameter.power 0.9924080       -1  0.05     40
10      brewtime parameter.power 0.9985188        1  0.05     40
11 I(brewtime^2) parameter.power 0.8331067        1  0.05     40
=======================Evaluation Info========================
• Alpha = 0.05 • Trials = 40 • Blocked = FALSE
• Evaluating Model = ~temp + roast + brewtime + I(brewtime ^ 2)
• Anticipated Coefficients = c(1, 1, 1, -1, 1, 1)
```

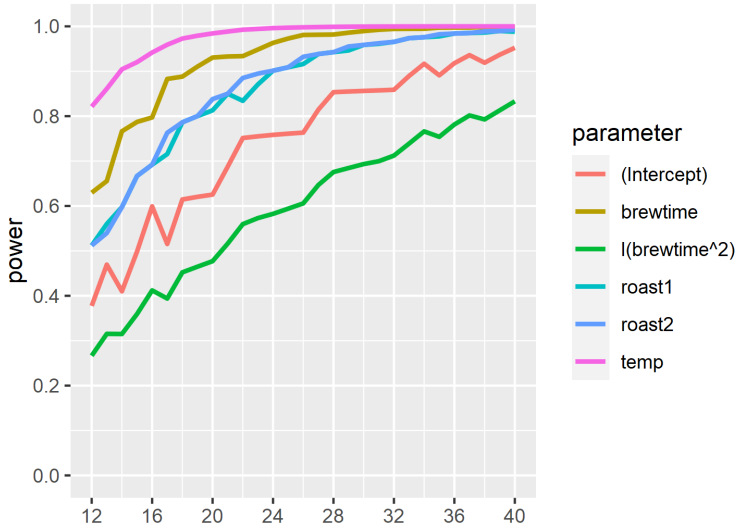


Figure 4: Parameter power versus sample size for all of the parameters in a design. If the user requires 80 percent power for their experiment, for anticipated coefficients of 1, it would take approximately 40 experimental runs to reach that benchmark in all factors for this model.

`gen_design()` also has an additional argument, `conservative`, that automatically generates anticipated coefficients for the most conservative estimate of effect power. For more information on conservative anticipated coefficients and to see how these calculations are performed, see Freeman, Johnson, and Simpson (2014) and Johnson, Medlin, Freeman, and Simpson (2019).

When calculating power for designs with hard-to-change factors, the situation is slightly different: statistical tests that do not consider the correlation between runs are biased (Ganju and Lucas 1997). OLS is no longer appropriate due to the introduction of run-to-run correlation, as OLS assumes complete independence between runs. We instead calculate our power using generalized least squares, which is a technique that allows estimating unknown parameters given intra-run correlation. The only difference in calculation is the insertion of the variance-covariance matrix V (which **skpr** calculates from a design's blocking structure and the `varianceratios` parameter) into the non-centrality parameter calculation, and a modification of the number of degrees of freedom for each term (see Pinheiro and Bates 2000, pp 91 for more details).

4. Evaluating power for designs with Monte Carlo methods

When designing an experiment with a non-normally distributed response variable, it can be difficult to calculate power analytically. Often, power for these designs is estimated by relying on a normal approximation to the underlying non-normal distribution (Johnson, Freeman, Simpson, and Anderson 2018). The validity of the normal approximation can be hard to quantify, and experiments with small sample sizes are where those approximations are likely to fail, potentially leading to inaccurate power calculations. In these cases, the best way to estimate the power is by simulating the experiment and calculating the power values via a Monte Carlo simulation. **skpr** provides a function, `eval_design_mc()`, that takes a design as

an input, performs a simulation of the experiment with user-defined simulation settings, and returns a data frame of power values for each parameter.

Monte Carlo techniques are a broad class of computational algorithms that use repeated random sampling to obtain numerical results. Calculating statistical power for a given design is a problem particularly well-suited for this simulation approach. This is due to two major benefits: first, the user can specify exactly how the data from the test is generated. This means that for a complex experimental design, a user can exactly specify how each factor influences the simulated response. Thus, the user has fine control over the effect size and number of effects contributing to the simulated outcome. Secondly, the power estimate can be generated using the same methods and techniques that are planned to be used when analyzing the real data from the experiment. If the user wants to fit a generalized linear model, they can calculate the power using the same libraries and methods they intend to use during the data analysis. The exact equivalence between the method used to generate the power and that used during the data analysis portion provides a strong analytic basis for the generated power values.

The algorithm used in **skpr** for calculating power via a Monte Carlo simulation is as follows:

1. Input the run matrix for the design being tested, the anticipated coefficients β, the model, the desired number of simulations N, and specify the function that generates the simulated output.
2. Generate the model matrix X using the run matrix and the model.
3. Repeat the following process N times:
   (a) Generate the simulated responses using $f(X, \beta)$.
   (b) Fit the design (using `lm` or `glm`).
   (c) Extract the p values for each term in the model and compare each one to the specified level set for α. If the p value for a term falls under that level, increment a counter specific to that term. This counts the total number of times that the term was significant in the N simulations.
4. For each term, divide the number of times the term was significant with the total number of simulations, N. This number is the estimated power.

The user-supplied function that generates simulated responses (argument `rfunction`), combines the anticipated coefficients and the model matrix to calculate the anticipated response. The following function is an example appropriate for a normal response variable with a standard deviation of one:

```
R> response = function(X, b) {
+    return(rnorm(n = nrow(X), mean= X %*% b, sd = 1))
+  }
```

This function generates N simulated responses (one for each experimental run). These responses are then fit using a linear model (if the family is Gaussian, as shown above), generalized linear model (for all other GLM families), survival model (`eval_design_survival_mc()`), or other user defined package (`eval_design_custom_mc()`).

Distribution	Blocking	Simulating function (`rfunction`)	Fit function
Gaussian	No	`rnorm(mean = X %*% b, sd = 1)`	`lm`
Gaussian	Yes	`rnorm(mean = X %*% b + d, sd = 1)`	`lmer`
Binomial	No	`rbinom(prob = 1/(1+exp(-(X %*% b))))`	`glm(family = "binomial")`
Binomial	Yes	`rbinom(prob = 1/(1+exp(-(X %*% b + d))))`	`glmer(family = "binomial")`
Poisson	No	`rpois(lambda = exp((X %*% b)))`	`glm(family = "poisson")`
Poisson	Yes	`rpois(lambda = exp((X %*% b + d)))`	`glmer(family = "poisson")`
Exponential	No	`rexp(rate = exp(-(X %*% b)))`	`glm(family = Gamma(link="log"))`
Exponential	Yes	`rexp(rate = exp(-(X %*% b + d)))`	`glmer(family = Gamma(link="log"))`

Table 1: A table of the distributions and random generating functions built-in to **skpr**. Each random generating function depends on the model matrix X, the anticipated coefficients b, and if there is a blocking term, an additional noise term d (for a random intercept term) associated with each block. Each function has a corresponding fitting function that is used to fit the model and generate p values for each term, which then are used to calculate power in the Monte Carlo simulation.

Function `eval_design_mc()` includes four built-in response generating functions for four distributions: Gaussian, Poisson, binomial, and exponential. When the user inputs a distribution into argument `glmfamily` (unless they specify their own simulation function), `eval_design_mc()` automatically uses one of the built-in functions. These are shown in Table 1. If the input is a split-plot design, the Monte Carlo method fits the non-blocking effects, treating the blocking factors as random intercepts. Modeling it in this manner improves the power for the non-blocking factors and more accurately represents the effect of hard-to-change factors.

Function `eval_design_mc()` also supports the calculation of effect power. **skpr** uses the `Anova` function from the **car** package (Fox and Weisberg 2011) to calculate the p values with a type-III sum of squares (SS) and with a Wald test. The user is able to change both the ANOVA SS type (II or III) and the default test from Wald to a likelihood-ratio, χ^2-, or F-test through the `advancedoptions` argument. If the user does not need these values, the effect power calculation can be turned off to speed up the Monte Carlo simulation.

The algorithm for calculating power for a blocked/split-plot design via a Monte Carlo simulation is as follows:

1. Input the run matrix for design being tested, the anticipated coefficients β, the model, the desired number of simulations N, the variance ratio(s) between split-plot strata, and specify the function $f(X, \beta)$ that generates the simulated output.
2. Calculate the model matrix X using the run matrix and the model.
3. Convert the row name structure into columns of block indicators.
4. Update the formula to include random effects for each of the blocking columns.
5. Repeat the following process N times:
   (a) If using a non-Gaussian generalized linear model, first run the steps (b)–(e) below with $\beta = 0$ to generate an empirical distribution of p values for the null distribution. This empirical distribution is then used to set the α cutoff for each parameter in the actual power calculation, to remove or reduce any type-I error inflation associated with the blocking.

(b) Generate a vector of block noise d, where each run in a block receives the same amount of noise.

(c) Generate the simulated responses using $f(X, \beta, d)$.

(d) Fit the design using **lmerTest** (Kuznetsova, Brockhoff, and Christensen 2017) or **lme4** (Bates, Mächler, Bolker, and Walker 2015).

(e) Extract the p values for each term in the model matrix and compare them to the specified level set for α (if a generalized linear model, this uses the empirical α for each parameter calculated in step (a)). For linear mixed models, this is done with the **lmerTest** package. If the p value for a term falls under that level, increment a counter specific to that term. This counts the total number of times that the term was significant in the N simulations.

6. For each term, divide the number of times the term was significant with the total number of simulations, N. This number is the power.

As an example, here we generate a design manually and add our own blocking columns. We evaluate the design with a Monte Carlo simulation both with and without blocking:

```
R> design <- data.frame(X1 = rep(c(1, -1), 6), X2 = c(rep(1, 6), rep(-1, 6)))
R> design[, "Block1"] <- c(1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 4)
R> design

   X1 X2 Block1
1   1  1      1
2  -1  1      1
3   1  1      1
4  -1  1      2
5   1  1      2
6  -1  1      2
7   1 -1      3
8  -1 -1      3
9   1 -1      3
10 -1 -1      4
11  1 -1      4
12 -1 -1      4

R> eval_design_mc(design, ~ X1 + X2, 0.05, calceffect = FALSE, nsim = 10000,
+    blocking = FALSE, parallel = TRUE)

    parameter               type  power
1 (Intercept) parameter.power.mc 0.8728
2          X1 parameter.power.mc 0.8637
3          X2 parameter.power.mc 0.8662
============Evaluation Info============
• Alpha = 0.05 • Trials = 12 • Blocked = FALSE
• Evaluating Model = ~ X1 + X2
• Anticipated Coefficients = c(1, 1, 1)
```

```
R> eval_design_mc(design, ~ X1 + X2, 0.05, calceffect = FALSE, nsim = 10000,
+    blocking = TRUE, parallel = TRUE)

    parameter               type  power
1 (Intercept) parameter.power.mc 0.1970
2          X1 parameter.power.mc 0.8384
3          X2 parameter.power.mc 0.1963
============Evaluation Info============
• Alpha = 0.05 • Trials = 12 • Blocked = TRUE
• Evaluating Model = ~ X1 + X2
• Anticipated Coefficients = c(1, 1, 1)
• Number of Blocks = Level 1: 4
• Variance Ratios  = Level 1: 1, Level 2: 1
```

When analyzed without blocking, the blocked column is detected and removed. However, when we turn on blocking, the blocking column is detected and a random effect is added to the model internally: `model = update(model, . + (1:Block1))`. Each blocking column adds an additional layer of nesting. For this design, the `X1` term is relatively unchanged in power since it is not correlated with the blocks. However, the `X2` term is correlated with the blocking term, and thus the power for that term is lower than in the unblocked case. Compared to a completely randomized design, split-plot designs generally have less statistical power for the hard-to-change factors (Ju and Lucas 2002).

For a binomial response, the user runs the additional risk of encountering partial separation (Lesaffre and Albert 1989), where the model is not able to fully fit some terms. `eval_design_mc()` does one final check at the end of the simulation process to check for the distribution of p values–if a spike is detected around $p = 1.00$, it is likely that there were repeated instances of partial separation occurring during the simulation. For well-behaved designs, the distribution of p values is flat or decreasing for increasing p values. `eval_design_mc()` splits the distribution of p values into bins of width 0.05, and compares the $0.95 - 1.00$ bin to the $0.80 - 0.85$, $0.85 - 0.90$, and $0.90 - 0.95$ bins. If the final bin is greater than the other three, `eval_design_mc()` displays a warning to the user that partial separation possibly has been detected. The user can choose to increase the number of runs, or implement their own evaluation routine with `eval_design_custom_mc()` that performs a penalized regression to allow the model to fit even when separation is present (see Section 4.2 for an example).

4.1. Evaluating Monte Carlo power for survival designs

Package **skpr** provides an interface through `eval_design_survival_mc()` to perform power analyses for experiments with left or right censored data using the **survival** package (Therneau 2021; Therneau and Grambsch 2000) in R. The main difference in the API as compared to `eval_design_mc()` is the random generation function `rfunction` needs to return a `survival::Surv` object that includes information on whether a point in that data set is censored. The point after (or before, for left censored data) which the data is censored in the simulation is given by the user as the argument `censorpoint`.

The algorithm **skpr** uses for calculating survival power via a Monte Carlo simulation is as follows:

1. Input the run matrix for design being tested, the anticipated coefficients β, the model, the desired number of simulations N, the censor type (left/right), the point before/after which the data is censored, and specify the function $f(X, \beta)$ that generates the simulated output.

2. Calculate the model matrix X using the design and the model.

3. Repeat the following process N times:

   (a) Generate the simulated responses using $f(X, \beta)$.

   (b) Censor the responses if they fall after (right censored) or before (left censored) the censor point.

   (c) Fit the design using the `survreg` function from the **survival** package.

   (d) Extract the p values for each term in the model matrix using `summary(model)` and compare them to the specified level set for α. If the p value for a term falls under that level, increment a counter specific to that term. This counts the total number of times that the term is significant in your N simulations.

4. For each term, divide the number of times each term was significant with the total number of simulations, N. This number is the power.

Function `eval_design_survival_mc()` has built-in support for Gaussian, exponential, and lognormal distributions, and users can supply their own random generating functions with the `rfunctionsurv` argument.

4.2. Custom Monte Carlo evaluation of power

In addition to generalized linear models and survival analyses, **skpr** also provides an interface for the user to supply their own libraries for power evaluation. The Monte Carlo algorithm needs methods to simulate responses, fit them, and extract p values from the resulting fit object. The algorithm is independent of the fitting object used, and needs a list of p values to compare to the set value for α to calculate power. The user passes these functions to `eval_design_custom_mc()` and the algorithm fits the data with the model provided and extracts the p values accordingly.

As an example, suppose a user wants to evaluate power for a design with a binomial response, and wishes to detect changes in probability from 0.5 to 0.8. There are two factors, each with two levels, and the user is including the interaction term. The user first generates an optimal design using the estimated budget of 40 runs, and then evaluates it in `eval_design_mc()`. We can see the numerical values of the anticipated coefficients generated by specifying `effectsize = c(0.5, 0.8)` for a binomial model in the evaluation information printed with the power values:

```
R> options(width = 75)
R> cand_set = expand.grid(X1 = c(1, -1), X2 = c(1, -1))
R> design_sep = gen_design(cand_set, ~ X1 * X2, 48)
R> eval_design_mc(design_sep, 0.2, glmfamily = "binomial",
+    effectsize = c(0.5, 0.8), nsim = 10000, calceffect = FALSE)
```

```
Warning in eval_design_mc(design_sep, 0.2, glmfamily = "binomial",
effectsize = c(0.5, : Partial or complete separation likely detected in the
binomial Monte Carlo simulation. Increase the number of runs in the design
or decrease the number of model parameters to improve power.

    parameter               type  power
1 (Intercept) parameter.power.mc 0.3123
2          X1 parameter.power.mc 0.3142
3          X2 parameter.power.mc 0.3142
4       X1:X2 parameter.power.mc 0.3075
============Evaluation Info============
• Alpha = 0.2 • Trials = 48 • Blocked = FALSE
• Evaluating Model = ~ X1 + X2 + X1:X2
• Anticipated Coefficients = c(0.693, 0.693, 0.693, 0.693)
```

`eval_design_mc()` shows the user a warning: in this design, partial or complete separation is detected in fitting the design. This means that the model was not able to be fit in at least some of the runs. The user can increase the number of runs to fix the issue, at the cost of a more expensive experiment—or use a penalized logistic regression. The user postulates that they do not need to add more runs if they instead change the method of analysis. Instead of a standard generalized linear model, the user fits a generalized linear model with a Firth correction (Firth 1992a; 1992b; 1993; Heinze and Schemper 2002). An R package that implements this is the **mbest** package (Perry 2016), which provides the function `firthglm()`. `eval_design_custom_mc()` has an argument `fitfunction` that allows us to specify our method of fitting. This requires a function to extract p values from the resulting fit object, as well as the random response generating function:

```
R> library("mbest")
R> fitfirth <- function(formula, X, contrastslist = NULL) {
+    return(glm(formula, data = X,  family = "binomial",
+      method = "firthglm.fit"))
+ }
R> pvalfirth <- function(fit) {
+    return(coef(summary(fit))[, 4])
+  }
R> rfunctionfirth <- function(X, b) {
+    return(rbinom(n = nrow(X), size = 1, prob = 1/(1 + exp(-(X %*% b)))))
+  }
```

These three functions are passed into `eval_design_custom_mc()` to perform the simulation and calculate power. With this function, the user only need specify the API to interact with the package being used, and **skpr** takes care of the power simulation, model matrix initialization, parallelization, as well as other programmatic scaffolding. Note that this function accepts only a length-one `effectsize` argument.

```
R> eval_design_custom_mc(design_sep, alpha = 0.2, nsim = 10000,
+    fitfunction = fitfirth, pvalfunction = pvalfirth,
+    rfunction = rfunctionfirth, effectsize = 2 * 0.6931472)
```

```
    parameter            type  power
1 (Intercept) custom.power.mc 0.6568
2          X1 custom.power.mc 0.6595
3          X2 custom.power.mc 0.6748
4       X1:X2 custom.power.mc 0.6521
==========Evaluation Info==========
• Alpha = 0.2 • Trials = 48 • Blocked = FALSE
• Evaluating Model = ~ X1 + X2 + X1:X2
• Anticipated Coefficients = c(0.693, 0.693, 0.693, 0.693)
```

The power is substantially improved due to fewer runs being penalized by the model not fitting at all. We can also check the type-I error rate (by setting `effectsize = 0`) and confirm that this is not the result of type-I error inflation, and that we do have a more powerful test.

```
R> eval_design_custom_mc(design_sep, alpha = 0.2, nsim = 10000,
+    fitfunction = fitfirth, pvalfunction = pvalfirth,
+    rfunction = rfunctionfirth, effectsize = 0)
```

```
    parameter            type  power
1 (Intercept) custom.power.mc 0.1965
2          X1 custom.power.mc 0.1927
3          X2 custom.power.mc 0.1995
4       X1:X2 custom.power.mc 0.1917
==========Evaluation Info==========
• Alpha = 0.2 • Trials = 48 • Blocked = FALSE
• Evaluating Model = ~ X1 + X2 + X1:X2
• Anticipated Coefficients = c(0, 0, 0, 0)
```

The type-I error rates are all below the acceptable rate of 0.2, showing that applying the Firth correction does result in a more powerful test. However, the power is still relatively low even after changing the analysis method, so the user concludes that they will have to add additional runs or remove terms from the model to increase it.

5. Graphical design evaluation

Power and optimality criteria are useful for design evaluation, but often practitioners have several competing goals when generating designs. A design that is "best" according to an optimality criterion might have poor prediction variance or unwanted correlations; to assess this, **skpr** provides two methods to assess a design graphically: fraction of design space plots, and correlation plots.

5.1. Fraction of design space plots

Function `plot_fds()` takes as an input a run matrix and a model, and plots a fraction of design space (FDS) plot for that design (Zahran, Anderson-Cook, and Myers 2003). A fraction of design space plot is a useful tool for assessing the scaled prediction variance across

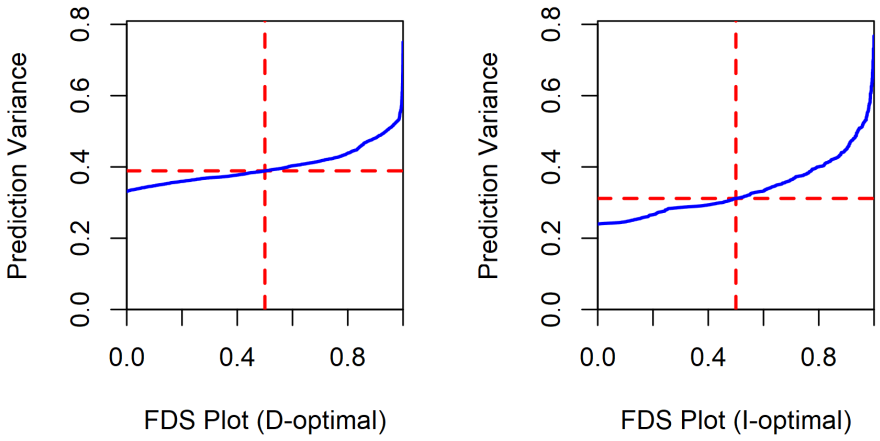


Figure 5: Fraction of design space plots in **skpr**, for two different 24-run designs with different optimality criteria. The plots show that the I-optimal design has lower prediction variance throughout most of the design space, but the D-optimal design has lower maximum prediction variance. The horizontal red line represents the median prediction variance. This plot was generated by passing a list of the two designs to the fraction of design space plotting function in **skpr**.

the design space (Goldfarb, Anderson-Cook, Borror, and Montgomery 2004). The FDS plot distills the design's prediction variance across the entire space into a single curve, making it a useful tool in assessing and comparing designs.

Other packages that generate FDS plots are **vdg** (Schoonees, le Roux, and Coetzer 2016), **VDgraph** (Lawson and Vining 2014) and **VdgRsm** (Srisuradetchai and Borkowski 2015). **skpr** provides its own implementation of FDS plots with `plot_fds()`. Unlike **VdgRsm**'s `fds.cube()`, there are no limits to the number of factors used in creating the plot. Like **vgd**, `plot_fds()` uses a Monte Carlo approach to generating fraction of design space plots. This method involves sampling randomly throughout the design space, calculating the prediction variance for each randomly chosen test point. The points are then ordered to generate a cumulative distribution function for the prediction variance as a function of the percentage of the design space. Rather than specifying the design itself, the user can pass into `plot_fds()` the output of one of **skpr**'s main functions–`gen_design()`, `eval_design()`, `eval_design_mc()`, or `eval_design_survival_mc()` and **skpr** automatically extracts the information needed to construct the plot.

FDS plots are most useful for comparing competing designs. As an example, the code below and Figure 5 show FDS plots for two 24-run response surface designs with identical candidate sets and models, but with different optimality criteria. The plots show that the I-optimal model yields substantially lower prediction variance throughout most of the design space, but the D-optimal design has a lower maximum prediction variance. In the example, we use the argument `continuouslength = 50` to generate higher-resolution FDS plots.

```
R> candset <- expand.grid(X1 = c(-1, 0, 1), X2 = c(-1, 0, 1),
+    X3 = as.factor(c("A", "B", "C")))
```

```
R> fds_design1 <- gen_design(candset, ~.*. + I(X1 ^ 2) + I(X2 ^ 2), 24,
+    optimality = "D")
R> fds_design2 <- gen_design(candset, ~.*. + I(X1 ^ 2) + I(X2 ^ 2), 24,
+    optimality = "I")
R> plot_fds(list(fds_design1,fds_design2), continuouslength = 50,
+    description = c("FDS Plot (D-optimal)","FDS Plot (I-optimal)"))
```

In general, we recommend only comparing FDS plots generated by **skpr** to other plots generated by **skpr**, because the curve will depend on the details of the sampling algorithm and the definition of "design space." For example, **skpr** assumes a hyper-cuboidal region for the design space, with continuous factors sampled uniformly between their maximum and minimum values, but another package might use a spheroidal region or a different sampling method.

5.2. Correlation plots

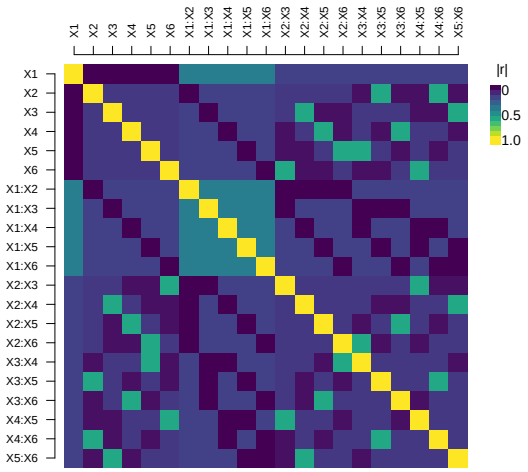
For non-full-factorial designs, correlation can often exist between main effects and the interaction terms that are not included in the model. This will introduce bias if those terms are actually active. Two designs can be equally optimal based on a criterion, but have different correlation structures due to terms not included in the original model (see Figure 1 for an example of this). Thus, it is good practice to examine the correlation structure of a design.

skpr provides the `plot_correlations()` function to display the absolute value of the correlation between any two effects. By default, `plot_correlations()` includes all of the main effects and all of the interactions, but the user can also input their own model or level of interactions they want to include and `plot_correlations()` builds a custom correlation map. The color scheme by default uses the **viridis** palette (Garnier 2021), but can also be customized by passing in a vector of color values to the `customcolors` argument.

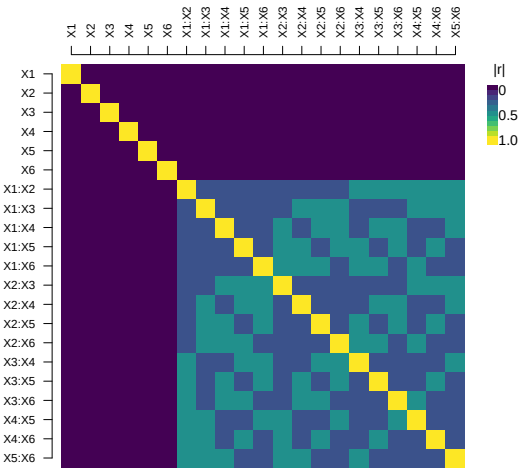
An illustrative example of the use of correlation plots involves comparing a 12-run Alias-optimal design with six continuous factors to a D-optimal design of the same inputs. The Alias-optimality criterion specifically minimizes correlation between main effect terms and interaction terms–thus, an Alias-optimal design should show no correlation between those pairs on the correlation plot. This comparison is shown in Figure 6. **skpr** by default chooses an orthonormal basis for generating contrasts with the built-in `contr.simplex()` function to make correlation plots display correctly. A full factorial design is an orthogonal design, but un-normalized sum contrasts show correlation between levels within a categorical factor (for categorical factors with more than two levels). Using an orthonormal basis correctly shows an orthogonal design as having no correlation between factor levels.

6. `skprGUI()`

skpr provides a graphical user interface, written with the **shiny** package (Chang *et al.* 2021) (and partly styled with the **shinythemes** package, Chang 2021), as a way of quickly generating and assessing multiple aspects of a design. Integrated into this tool are several diagnostic plots to help evaluate designs, as well as most of the major functionality included in the package. The interface is meant to improve ease of use and access–this application can be deployed on a web server and accessed and used either locally or over the internet. This allows users without any knowledge of R to access the capabilities provided by **skpr** without having to learn how to



(a) Correlation matrix for a D-optimal design.



(b) Correlation matrix for an Alias-optimal design.

Parameter	Power
(Intercept)	0.969
X1	0.969
X2	0.969
X3	0.969
X4	0.969
X5	0.969
X6	0.969

(c) Power values for D-optimal design.

Parameter	Power
(Intercept)	0.945
X1	0.969
X2	0.945
X3	0.945
X4	0.945
X5	0.945
X6	0.945

(d) Power values for Alias-optimal design.

Figure 6: Correlation matrices and calculated power for two designs with the same input parameters, but different optimality criteria. The Alias-optimal design does not have any correlation between main effect terms and two-factor interactions. However, that nicer correlation structure comes at the expense of statistical power (calculated with `effectsize = 2` and `alpha = 0.2`). Depending on the type of the experiment, the practitioner might value one property over the other. This type of trade-off is why the practitioner should use multiple methods to analyze and evaluate a design.

install and use R. A walk through of the interface is built-in to the **shiny** application using the **rintrojs** package (Ganz 2016) via the "Tutorial" button. The user interface of the application, shown in Figure 7, has the following tabs:

1. Basic: The inputs required for design generation, such as factor level settings, the number of runs, and the model. The user can also specify whether a factor is easy or hard-to-change. If the user makes a factor hard-to-change, an additional input pane appears that allows the user to specify the number of blocks. **skpr** will allocate the sub plots as evenly as possible between the whole-plots, given the number of trials.

2. Advanced: Optional arguments for the user to customize design generation and evaluation. In this pane, the user can specify the optimality criterion, number of repeat searches when generating a design, the variance ratio between whole-plots and sub-plots,

set a seed for random number generation, toggle multicore search, and turn on advanced evaluation options. This is also where the user can specify the minimum D-optimality and degree of interactions used when generating Alias-optimal designs.

3. Power: This pane allows the user to select the type of power analysis they want to run when they evaluate the design, as well as specify the effect size and their acceptable type-I error rate α. The type option corresponds to the different evaluation functions provided in **skpr**: "Linear Model" calculates power with `eval_design()`, "Generalized Linear Model" calculates power using `eval_design_mc()`, and ''Survival Model" calculates power using `eval_design_survival_mc()`. If the user chooses to evaluate using a generalized linear model or a survival model, they can also pick the number of Monte Carlo simulations as well as the distributional family of the response variable. The effect size option also changes according to distribution: the binomial effect size selector is given as a range of two probabilities, while the exponential and Poisson effect sizes are expressed in terms of two different means/expected event numbers (respectively). Finally, the survival optional also allows the user to specify at which point the data is censored in the simulation, as well as the type of censoring (left or right).

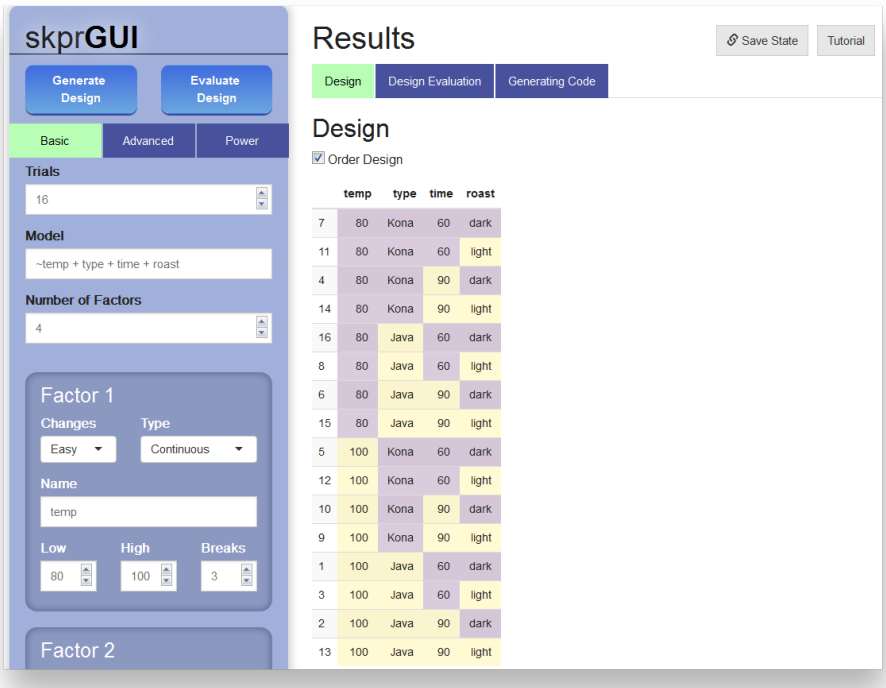
The output panes display the following information:

1. Design: This is a color-coded matrix showing the generated design's run matrix. By default the design is randomized, but the user can order the design from left to right by clicking the "Order Design" check box.

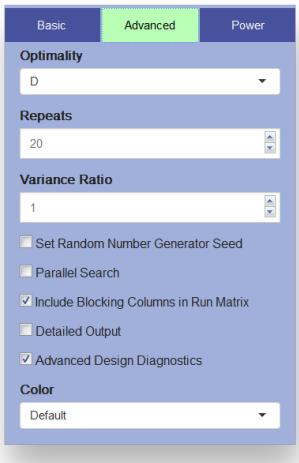
2. Design Evaluation: This pane displays all the design diagnostics computed by **skpr**: the power calculation, correlation plot, fraction of design space plot, optimality criteria values, and optimal search values (see Figure 8). If the user is running a Monte Carlo simulation, it displays three additional figures: The simulated response estimates (which are the actual estimated values of the response from the simulation), the parameter estimates for each model term, and the simulated p values.

3. Generating Code: The GUI provides a code completion pane (see Figure 9) that eases a user's transition from the GUI to code by showing the exact lines of code used to generate and evaluate a design. This code pane updates in real time as the user changes the model, factors, number of runs, etc. The user can then use this code to either save the work they performed in the GUI, or use that code as a base to springboard into more complex analyses. In addition to helping transition users to scripting, this code also shows the user how the power analysis they performed corresponds to the methods used to analyze the experiment once it is completed. This helps bridge the conceptual gap between the pre-experiment power calculation and the post-experiment final analysis. While most of the functionality of **skpr** is built-in to the GUI, most of the advanced customization options can only be accessed through the library functions (e.g., `eval_design_custom_mc()`). An example of different designs (split-plot versus non-split-plot) and their resulting different methods of analysis displayed in the code pane is shown in Figure 9.

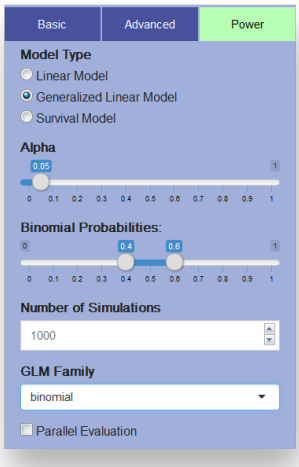
There are three versions of the **shiny** application provided, depending on how the user wants to deploy the application. `skprGUI()` opens up a widget within the RStudio IDE (**RStudio**



(a) Basic inputs pane along with the design pane.



(b) Advanced inputs pane.



(c) Power inputs pane.

Figure 7: (a) Basic input pane in `skprGUI()` along with the design pane. The input pane allows the users to customize their experimental factors, design generation settings, and power evaluation options. In the design pane, the factor levels are colored in each column independently by level. The design can be randomized or de-randomized with the "Order Design" check box. (b) Advanced options: This is where the user can change optimality criterion and customize advanced search options. (c) Power: This pane allows the user to select the type of power analysis, as well as specify effect size and the allowable type-I error rate. For survival analysis, this is also where the user can select the censoring type and censor point.

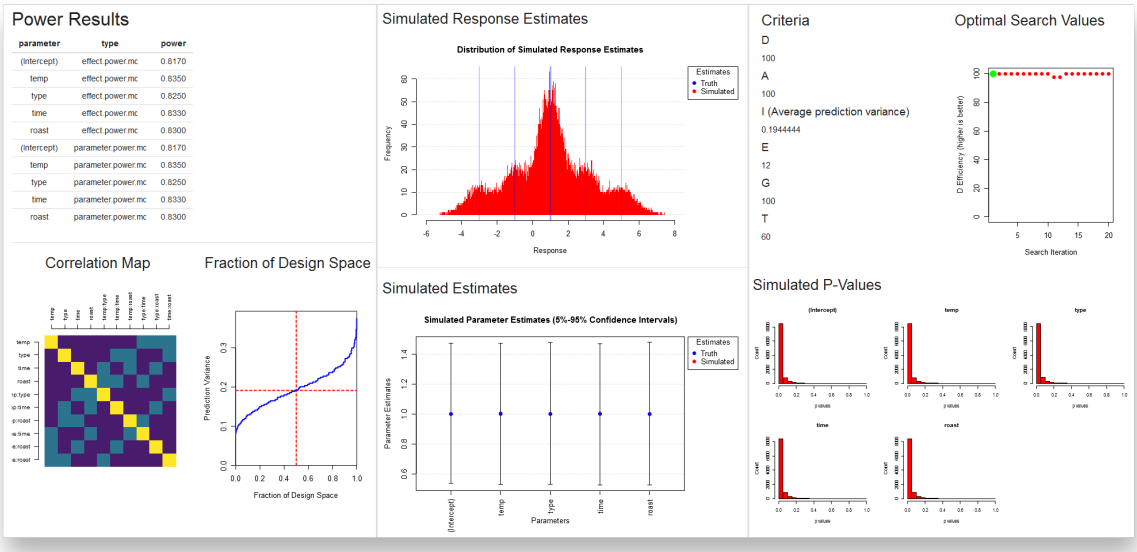


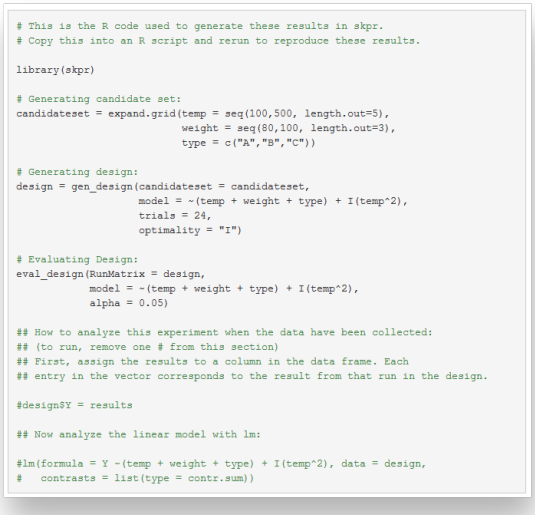
Figure 8: Design evaluation pane in `skprGUI()`. The evaluation pane displays the power calculation, correlation plot, fraction of design space plot, optimality criteria values, and optimal search values by default. The optimal search values can be used to judge the breadth of the search; an "optimal" design that is found only once during the search process is much more likely to be locally optimal than globally optimal, compared to a design that is found multiple times. If the user is running a Monte Carlo simulation to calculate power, `skprGUI()` displays additional diagnostic plots showing the simulated response estimates and the simulated parameter estimates as well.

Team 2021), while `skprGUIbrowser()` opens up an external browser. `skprGUIserver()` is a version of the GUI that is meant to be deployed on a server–it disables the multicore option and integrates the **future** (Bengtsson 2020) and **promises** (Cheng 2021) packages to provide asynchronous support to better support multiple users at the same time. Otherwise, there are no differences between the three options.

7. Conclusions

skpr is an open-source R package that provides both optimal design generation and power evaluation tools. It improves upon existing open-source algorithmic design generation packages in R by providing a wider variety of design generation options and improved search algorithms, as well as power evaluation facilities, plotting options, and an optional GUI. The package includes fast and efficient routines for design generation, allowing the user to quickly create an experiment best-suited for their particular purpose. **skpr** provides a general optimal design generation framework by providing the user a diverse set of built-in optimality criteria, as well as the option to use user-supplied criteria. It can evaluate the power of designs both analytically (for normal response variables) and via Monte-Carlo simulation.

The inclusion of the graphical user interface allows users to quickly prototype and generate designs, and extends the capabilities provided by **skpr** to those who do not normally use the R programming language. The ability to bookmark and save analyses done in the GUI ensure



```
# This is the R code used to generate these results in skpr.
# Copy this into an R script and rerun to reproduce these results.

library(skpr)

# Generating candidate set:
candidateset = expand.grid(temp = seq(100,500, length.out=5),
                           weight = seq(80,100, length.out=3),
                           type = c("A","B","C"))

# Generating design:
design = gen_design(candidateset = candidateset,
                    model = ~(temp + weight + type) + I(temp^2),
                    trials = 24,
                    optimality = "I")

# Evaluating Design:
eval_design(RunMatrix = design,
            model = ~(temp + weight + type) + I(temp^2),
            alpha = 0.05)

## How to analyze this experiment when the data have been collected:
## (to run, remove one # from this section)
## First, assign the results to a column in the data frame. Each
## entry in the vector corresponds to the result from that run in the design.

#design$Y = results

## Now analyze the linear model with lm:

#lm(formula = Y ~(temp + weight + type) + I(temp^2), data = design,
#   contrasts = list(type = contr.sum))
```

(a) Example code pane for non-split-plot design.

```
# This is the R code used to generate these results in skpr.
# Copy this into an R script and rerun to reproduce these results.

library(skpr)

# Generating candidate set:
candidateset = expand.grid(temp = seq(100,500, length.out=5),
                           weight = seq(80,100, length.out=3),
                           type = c("A","B","C"))

# Generating design for hard-to-change factors:
design_htc = gen_design(candidateset = candidateset,
                        model = ~temp,
                        trials = 4)

# Generating design:
design = gen_design(candidateset = candidateset,
                    model = ~(temp + weight + type),
                    trials = 24,
                    splitplotdesign = design_htc,
                    splitplotsizes = 6,
                    splitcolumns = TRUE)

# Evaluating Design:
eval_design(RunMatrix = design,
            model = ~(temp + weight + type),
            alpha = 0.05,
            blocking = TRUE)

## How to analyze this experiment when the data have been collected:
## (to run, remove one # from this section)
## First, assign the results to a column in the data frame. Each
## entry in the vector corresponds to the result from that run in the design.

#design$Y = results

## Now analyze the blocked linear model with lmer (from the lme4 package):

#lme4::lmer(formula = Y ~ (1|Block1) + (temp + weight + type), data = design,
#          contrasts = list(type = contr.sum)))
```

(b) Example code pane for split-plot design.

Figure 9: `skprGUI()` provides the R code used to call **skpr**'s functions to recreate the actions performed in the GUI in a script. Part a) shows the code pane for a design without any split-plot structure, while part b) shows the same inputs, but with a hard-to-change factor included. Note how the code changes to account for the blocking, and how the analysis section changes from `lm` to `lmer` and the model includes a random effect. The displayed code updates in real time with changes made to the inputs. It also shows the user with comments and example code how their power analysis corresponds to the planned final analysis. This tool is meant to ease users familiar with a GUI interface into writing code, improve reproducibility, and provide a conceptual link between power and actual data analysis.

that these analyses are reproducible, and can be saved and shared. By providing a wide suite of evaluation and diagnostic tools, the GUI allows the user to explore and evaluate the design as a whole, rather than on a single criterion.

References

Aitken AC (1936). "IV.–On Least Squares and Linear Combination of Observations." *Proceedings of the Royal Society of Edinburgh*, **55**, 42–48. doi:10.1017/s0370164600014346.

Atkinson AC, Donev AN (1992). *Optimum Experimental Designs*. Oxford Science Publications. Clarendon Press.

Bates D, Eddelbuettel D (2013). "Fast and Elegant Numerical Linear Algebra Using the **RcppEigen** Package." *Journal of Statistical Software*, **52**(5), 1–24. doi:10.18637/jss.v052.i05.

Bates D, Mächler M, Bolker B, Walker S (2015). "Fitting Linear Mixed-Effects Models Using **lme4**." *Journal of Statistical Software*, **67**(1), 1–48. doi:10.18637/jss.v067.i01.

Bengtsson H (2020). ***future***: *Unified Parallel and Distributed Processing in R for Everyone*. R package version 1.21.0, URL https://CRAN.R-project.org/package=future.

Champely S (2020). ***pwr***: *Basic Functions for Power Analysis*. R package version 1.3-0, URL https://CRAN.R-project.org/package=pwr.

Chang W (2021). ***shinythemes***: *Themes for* ***shiny***. R package version 1.2.0, URL https://CRAN.R-project.org/package=shinythemes.

Chang W, Cheng J, Allaire JJ, Sievert C, Schloerke B, Xie Y, Allen J, McPherson J, Dipert A, Borges B (2021). ***shiny***: *Web Application Framework for R*. R package version 1.6.0, URL https://CRAN.R-project.org/package=shiny.

Cheng J (2021). ***promises***: *Abstractions for Promise-Based Asynchronous Programming*. R package version 1.2.0.1, URL https://CRAN.R-project.org/package=promises.

Cook RD, Nachtrheim CJ (1980). "A Comparison of Algorithms for Constructing Exact D-Optimal Designs." *Technometrics*, **22**(3), 315–324. doi:10.1080/00401706.1980.10486162.

Cornell JA (1973). "Experiments with Mixtures: A Review." *Technometrics*, **15**(3), 437–455. doi:10.1080/00401706.1973.10489071.

Eddelbuettel D (2013). *Seamless R and C++ Integration with* ***Rcpp***. Springer-Verlag, New York. doi:10.1007/978-1-4614-6868-4.

Faul F, Erdfelder E, Lang AG, Buchner A (2007). "**G*Power** 3: A Flexible Statistical Power Analysis Program for the Social, Behavioral, and Biomedical Sciences." *Behavior Research Methods*, **39**(2), 175–191. doi:10.3758/bf03193146.

Firth D (1992a). "Bias Reduction, the Jeffreys Prior and GLIM." In L Fahrmeir, B Francis, R Gilchrist, G Tutz (eds.), *Advances in GLIM and Statistical Modelling*, pp. 91–100. Springer-Verlag, New York.

Firth D (1992b). "Generalized Linear Models and Jeffreys Priors: An Iterative Weighted Least-Squares Approach." In Y Dodge, J Whittaker (eds.), *Computational Statistics*, pp. 553–557. Physica-Verlag HD, Heidelberg.

Firth D (1993). "Bias Reduction of Maximum Likelihood Estimates." *Biometrika*, **80**(1), 27–38. doi:10.2307/2336755.

Fisher RA (1925). *Statistical Methods for Research Workers*. Oliver and Boyd, Edinburgh.

Fox J (2005). "The R Commander: A Basic Statistics Graphical User Interface to R." *Journal of Statistical Software*, **14**(9), 1–42. doi:10.18637/jss.v014.i09.

Fox J, Weisberg S (2011). *An R Companion to Applied Regression*. 2nd edition. Sage Publications, Thousand Oaks. URL http://socserv.socsci.mcmaster.ca/jfox/Books/Companion.

Freeman LJ, Johnson TH, Simpson JR (2014). "Power Analysis Tutorial for Experimental Design Software." *Technical report*, Institute for Defense Analyses. URL https://apps.dtic.mil/sti/pdfs/ADA619843.pdf.

Ganju J, Lucas JM (1997). "Bias in Test Statistics When Restrictions in Randomization Are Caused by Factors." *Communications in Statistics – Theory and Methods*, **26**(1), 47–63. doi:10.1080/03610929708831901.

Ganz C (2016). "**rintrojs**: A Wrapper for the **Intro.js** Library." *Journal of Open Source Software*, **1**(6). doi:10.21105/joss.00063.

Garnier S (2021). ***viridis***: *Default Color Maps from* ***matplotlib***. R package version 0.6.1, URL https://CRAN.R-project.org/package=viridis.

Gaujoux R (2020). ***doRNG***: *Generic Reproducible Parallel Backend for* `foreach` *Loops*. R package version 1.8.2, URL https://CRAN.R-project.org/package=doRNG.

Goldfarb HB, Anderson-Cook C, Borror CM, Montgomery DC (2004). "Fraction of Design Space Plots for Assessing Mixture and Mixture-Process Designs." *Journal of Quality Technology*, **36**(2), 169–179. doi:10.1080/00224065.2004.11980263.

Goos P (2002). *The Optimal Design of Blocked and Split-Plot Experiments*. Lecture Notes in Statistics. Springer-Verlag.

Goos P (2005). "The Usefulness of Optimal Design for Generating Blocked and Split-Plot Response Surface Experiments." *Working Papers 2005033*, University of Antwerp, Faculty of Applied Economics. URL https://ideas.repec.org/p/ant/wpaper/2005033.html.

Goos P, Jones B (2011). *Optimal Design of Experiments: A Case Study Approach*. John Wiley & Sons.

Goos P, Vandebroek M (2001). "D-Optimal Response Surface Designs in the Presence of Random Block Effects." *Computational Statistics & Data Analysis*, **37**(4), 433 – 453. doi:10.1016/s0167-9473(01)00010-x.

Goos P, Vandebroek M (2003). "D-Optimal Split-Plot Designs with Given Numbers and Sizes of Whole Plots." *Technometrics*, **45**(3), 235–245. doi:10.1198/004017003000000050.

Goos P, Vandebroek M (2004). "Outperforming Completely Randomized Designs." *Journal of Quality Technology*, **36**(1), 12–26. doi:10.1080/00224065.2004.11980249.

Green P, MacLeod CJ (2016). "**SIMR**: An R Package for Power Analysis of Generalized Linear Mixed Models by Simulation." *Methods in Ecology and Evolution*, **7**(4), 493–498. doi:10.1111/2041-210x.12504.

Grömping U (2013). ***FrF2.catlg128***: *Catalogues of Resolution IV 128 Run 2-Level Fractional Factorials Up to 33 Factors That Do Have 5-Letter Words*. R package version 1.2-1, URL https://CRAN.R-project.org/package=FrF2.catlg128.

Grömping U (2014a). ***RcmdrPlugin.DoE***: *R Commander Plugin for (Industrial) Design of Experiments*. R package version 0.12-3, URL https://CRAN.R-project.org/package=RcmdrPlugin.DoE.

Grömping U (2014b). "R Package **FrF2** for Creating and Analyzing Fractional Factorial 2-Level Designs." *Journal of Statistical Software*, **56**(1), 1–56. URL http://www.jstatsoft.org/v56/i01/.

Grömping U (2018). "R Package **DoE.base** for Factorial Experiments." *Journal of Statistical Software*, **85**(5), 1–41. doi:10.18637/jss.v085.i05.

Grömping U (2020). ***DoE.wrapper**: Wrapper Package for Design of Experiments Functionality*. R package version 0.11, URL https://CRAN.R-project.org/package=DoE.wrapper.

Gurobi Optimization, LLC (2019). ***Gurobi** Optimizer Reference Manual*. URL http://www.Gurobi.com/.

Harman R, Filova L (2019). ***OptimalDesign**: A Toolbox for Computing Efficient Designs of Experiments*. R package version 1.0.1, URL https://CRAN.R-project.org/package=OptimalDesign.

Heinze G, Schemper M (2002). "A Solution to the Problem of Separation in Logistic Regression." *Statistics in Medicine*, **21**(16), 2409–2419. doi:10.1002/sim.1047.

Johnson TH, Freeman L, Simpson J, Anderson C (2018). "Power Approximations for Generalized Linear Models Using the Signal-to-Noise Transformation Method." *Quality Engineering*, **30**(3), 511–524. doi:10.1080/08982112.2017.1361537.

Johnson TH, Medlin RM, Freeman LJ, Simpson JR (2019). "On Scoping a Test That Addresses the Wrong Objective." *Quality Engineering*, **31**(2), 230–239. doi:10.1080/08982112.2018.1479035.

Jones B, Nachtsheim CJ (2009). "Split-Plot Designs: What, Why, and How." *Journal of Quality Technology*, **41**(4), 340–361. doi:10.1080/00224065.2009.11917790.

Jones B, Nachtsheim CJ (2011a). "A Class of Three-Level Designs for Definitive Screening in the Presence of Second-Order Effects." *Journal of Quality Technology*, **43**(1), 1–15. doi:10.1080/00224065.2011.11917841.

Jones B, Nachtsheim CJ (2011b). "Efficient Designs with Minimal Aliasing." *Technometrics*, **53**(1), 62–71. doi:10.1198/tech.2010.09113.

Ju HL, Lucas JM (2002). "L^k Factorial Experiments with Hard-to-Change and Easy-to-Change Factors." *Journal of Quality Technology*, **34**, 411–421. doi:10.1080/00224065.2002.11980173.

Kane MJ, Emerson J, Weston S (2013). "Scalable Strategies for Computing with Massive Data." *Journal of Statistical Software*, **55**(14), 1–19. doi:10.18637/jss.v055.i14.

Kobilinsky A, Bouvier A, Monod H (2020). ***planor**: An R Package for the Automatic Generation of Regular Fractional Factorial Designs*. R package version 1.5-3, URL https://CRAN.R-project.org/src/contrib/Archive/planor/.

Kuznetsova A, Brockhoff PB, Christensen RHB (2017). "**lmerTest** Package: Tests in Linear Mixed Effects Models." *Journal of Statistical Software*, **82**(13), 1–26. doi:10.18637/jss.v082.i13.

Lawson J, Vining G (2014). ***Vdgraph**: Variance Dispersion Graphs and Fraction of Design Space Plots for Response Surface Designs*. R package version 2.2-2, URL https://CRAN.R-project.org/package=Vdgraph.

Lenth RV (2009). "Response-Surface Methods in R, Using **rsm**." *Journal of Statistical Software*, **32**(7), 1–17. doi:10.18637/jss.v032.i07.

Lesaffre E, Albert A (1989). "Partial Separation in Logistic Discrimination." *Journal of the Royal Statistical Society B*, **51**(1), 109–116. doi:10.1111/j.2517-6161.1989.tb01752.x.

Letsinger JD, Myers RH, Lentner M (1996). "Response Surface Methods for Bi-Randomization Structures." *Journal of Quality Technology*, **28**(4), 381–397. doi:10.1080/00224065.1996.11979697.

Microsoft, Weston S (2020). ***foreach**: Provides Foreach Looping Construct for R*. R package version 1.5.1, URL https://CRAN.R-project.org/package=foreach.

Morgan-Wall T, Khoury G (2021). ***skpr**: Design of Experiments Suite: Generate and Evaluate Optimal Designs*. R package version 1.0.0, URL https://CRAN.R-project.org/package=skpr.

Nachtsheim CJ, Jones B (2018). "Design Augmentation for Response Optimization and Model Estimation." *Quality Engineering*, **30**(1), 38–51. doi:10.1080/08982112.2017.1382298.

Perry PO (2016). "Fast Moment-Based Estimation for Hierarchical Models." *Journal of the Royal Statistical Society B*, **79**(1), 267–291. doi:10.1111/rssb.12165.

Pinheiro JC, Bates DM (2000). *Mixed-Effects Models in S and S-PLUS*. Statistics and Computing. Springer-Verlag.

R Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

RStudio Team (2021). ***RStudio**: Integrated Development Environment for R*. RStudio, PBC, Boston. URL http://www.RStudio.com/.

SAS Institute Inc (2018). ***JMP**, Version 14*. Cary. URL https://www.JMP.com/.

Scheffé H (1958). "Experiments with Mixtures." *Journal of the Royal Statistical Society B*, **20**(2), 344–360. URL http://www.jstor.org/stable/2983895.

Schoonees P, le Roux N, Coetzer R (2016). "Flexible Graphical Assessment of Experimental Designs in R: The **vdg** Package." *Journal of Statistical Software*, **74**(3), 1–22. doi:10.18637/jss.v074.i03.

Srisuradetchai P, Borkowski JJ (2015). ***VdgRsm**: Plots of Scaled Prediction Variances for Response Surface Designs*. R package version 1.5, URL https://CRAN.R-project.org/package=VdgRsm.

Stat-Ease, Inc (2021). ***Design-Expert** Software*. Minneapolis. Version 13, URL https://www.statease.com/software/design-expert/.

Studden WJ (1977). "Optimal Designs for Integrated Variance in Polynomial Regression." In *Statistical Decision Theory and Related Topics*, pp. 411–420. Academic Press. doi:10.1016/b978-0-12-307560-4.50026-0.

Therneau TM (2021). ***survival***: *Survival Analysis*. R package version 3.2-12, URL https://CRAN.R-project.org/package=survival.

Therneau TM, Grambsch PM (2000). *Modeling Survival Data: Extending the Cox Model*. Springer-Verlag.

Venables B (2013). ***conf.design***: *Construction of Factorial Designs*. R package version 2.0.0, URL https://CRAN.R-project.org/package=conf.design.

Wheeler B (2019). ***AlgDesign***: *Algorithmic Experimental Design*. R package version 1.2.0, URL https://CRAN.R-project.org/package=AlgDesign.

Wickham H (2014). "Tidy Data." *Journal of Statistical Software*, **59**(10), 1–23. doi:10.18637/jss.v059.i10.

Yates F (1935). "Complex Experiments." *Supplement to the Journal of the Royal Statistical Society*, **2**(2), 181–247. doi:10.2307/2983638.

Zahran A, Anderson-Cook C, Myers RH (2003). "Fraction of Design Space to Assess the Prediction Capability of Response Surface Designs." *Journal of Quality Technology*, **35**, 377–386. doi:10.1080/00224065.2003.11980235.

Affiliation:

Tyler Morgan-Wall
Institute for Defense Analyses
4850 Mark Center Drive
Alexandria, VA 22311, United States of America
E-mail: tmorganw@ida.org

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
August 2021, Volume 99, Issue 1 *Submitted:* 2019-08-12
doi:10.18637/jss.v099.i01 *Accepted:* 2020-11-25