Package 'glmmrOptim'

September 11, 2023

Type Package

Title Approximate Optimal Experimental Designs Using Generalised Linear Mixed Models

Version 0.3.2

Date 2023-09-11

Maintainer Sam Watson <S.I.Watson@bham.ac.uk>

Description Optimal design analysis algorithms for any study design that can be represented or modelled as a generalised linear mixed model including cluster randomised trials, cohort studies, spatial and temporal epidemiological studies, and split-plot designs. See <https://github.com/samuel-watson/glmmrBase/blob/master/README.md> for a detailed manual on model specification. A detailed discussion of the methods in this package can be found in Watson and Pan (2022) <arXiv:2207.09183>.

License GPL (>= 2)

Imports methods, Rcpp (>= 1.0.7), digest

LinkingTo Rcpp (>= 1.0.7), RcppEigen, RcppProgress, glmmrBase (>= 0.4.5), SparseChol (>= 0.2.1), BH, rminqa (>= 0.2.2)

RoxygenNote 7.2.3

NeedsCompilation yes

Author Sam Watson [aut, cre], Yi Pan [aut]

URL https://github.com/samuel-watson/glmmrOptim

BugReports https://github.com/samuel-watson/glmmrOptim/issues

Suggests testthat, CVXR

Biarch true

Depends R (>= 3.4.0), Matrix, glmmrBase

SystemRequirements GNU make

Encoding UTF-8

Repository CRAN

Date/Publication 2023-09-11 10:40:02 UTC

R topics documented:

apportion 2
DesignSpace 3
setParallelOptim 11

Index **13**

`apportion` *Generate exact designs from approximate weights*

Description

Given a set of optimal weights for experimental conditions generate exact designs using several rounding methods.

Usage

```
apportion(w, n)
```

Arguments

`w`	A vector of weights.
`n`	The size of the exact designs to return.

Details

Allocating ‘n‘ items to ‘k‘ groups proportionally to set of weights ‘w‘ is known as the apportionment problem. The problem most famously arose when determining how many members each state should have in the U.S. House of Representatives based on their proportion of the population. The solutions are named after their proposers in the early U.S. Hamilton’s method initially allocates ‘floor(n*w)‘ observations to each experimental condition and then allocates the remaining observations based on the largest remainders ‘n*w - floor(n*w)‘. The other methods (Adams, Jefferson, and Webster) are divisor methods. The vector of counts is ‘m‘, which is either all zeros for Jefferson and Webster and all ones for Adams, and we define ‘pi <- n*w‘ and then iteratively add observations based on the largest values of ‘pi/alpha‘ where ‘alpha‘ is either: * m + 0.5 (Webster) * m + 1 (Jefferson) * m (Adams) Pukelsheim and Rieder, 1996 <doi:10.2307/2337232> discuss efficient rounding of experimental condition weights and determine that a variant of Adam’s method is the most efficient. Results using this method are labelled "Pukelsheim" in the output; there may be multiple designs using this procedure. Pukelsheim and Rieder’s method assumes there is a minimum of one experimental condition of each type, whereas the other methods do not have this restriction.

Value

A named list. The names correspond to the method of rounding (see Details), and the entries are vectors of integers indicating the count of each type of experimental condition.

Examples

```
w <- c(0.45,0.03,0.02,0.02,0.03,0.45)
apportion(w,10)
```

`DesignSpace` *A GLMM Design Space*

Description

A GLMM Design Space

A GLMM Design Space

Details

A class-based representation of a "design space" that contains one or more Model objects.

An experimental study is comprised of a collection of experimental conditions, which are one or more observations made a pre-specified locations/values of covariates. A design space represents the collection of all possible experimental conditions for the study design and plausible models describing the data generating process. The main purpose of this class is to identify optimal study designs, that is the set of ‘n‘ experimental conditions from all possible experimental conditions that minimise the variance of a parameter of interest across the specified GLMMs.

A ‘DesignSpace‘ object is intialised using one or more Model objects. Design objects can be added or removed from the collection. All designs must have the same number of rows in their design matrices (X and Z) and the same number of experimental conditions. The DesignSpace functions can modify the linked design objects.

Initialisation The experimental condition refers to the smallest "unit" of the study design that could be included in the design. For example, in a cluster randomised trial, the experimental condition may be single individuals such that we can observed any number of individuals in any cluster period (including none at all). In this case the experimental condition would be equivalent to row number. Alternatively, we may have to observe whole cluster periods, and we need to choose which cluster periods to observe, in which case the each observation in a different cluster-period would have the same experimental condition identifier. Finally, we may determine that the whole cluster in all periods (a "sequence") is either observed or not.

Approximate c-Optimal designs The function returns approximate c-optimal design(s) of size m from the design space with N experimental conditions. The objective function is

$$C^T M^{-1} C$$

where M is the information matrix and C is a vector. Typically C will be a vector of zeros with a single 1 in the position of the parameter of interest. For example, if the columns of X in the design are an interdept, the treatment indicator, and then time period indicators, the vector C may be ‘c(0,1,0,0,...)‘, such that the objective function is the variance of that parameter. If there are multiple designs in the design space, the C vectors do not have to be the same as the columns of X in each design might differ.

If the experimental conditions are correlated with one another, then one of three combinatorial algorithms can be used, see Watson and Pan, 2022 <arXiv:2207.09183>. The algorithms are: (i) local search, which starts from a random design of size m and then makes the best swap between an experimental condition in and out of the design until no variance improving swap can be made; (ii) greedy search, which starts from a design of size p « n and then sequentially adds the best experimental condition until it generates a design of size m; (iii) reverse greedy search, which starts from the complete set of N experimental conditions and sequentially removes the worst experimental condition until it generates a design of size m. Note that only the local search has provable bounds on the solution.

If the experimental conditional are uncorrelated (but there is correlation between observations within the same experimental condition) then optionally an alternative algorithm can be used to approximate the optimal design using a second-order cone program (see Sangol, 2015 <doi:10.1016/j.jspi.2010.11.031> and Holland-Letz et al 2011 <doi:10.1111/j.1467-9868.2010.00757.x>). The approximate algorithm will return weights in [0,1] for each unique experimental condition representing the "proportion of effort" to spend on each design condition. There are different ways to translate these weights into integer values, which are returned see apportion. Use of the approximate optimal design algorithm can be disabled used ‘use_combin=TRUE‘

In some cases the optimal design will not be full rank with respect to the design matrix X of the design space. This will result in a non-positive definite information matrix, and an error. The program will indicate which columns of X are likely "empty" in the optimal design. The user can then optionally remove these columns in the algorithm using the ‘rm_cols‘ argument, which will delete the specified columns and linked observations before starting the algorithm.

The algorithm will also identify robust optimal designs if there are multiple designs in the design space. There are two options for robust optimisation. Both involve a weighted combination of the value of the function from each design, where the weights are specified by the ‘weights‘ field in the design space. The weights represent the prior probability or plausibility of each design. The weighted sum is either a sum of the absolute value of the c-optimal criterion or its log (e.g. see Dette, 1993 <doi:10.1214/aos/1176349149>).

Value

An environment that is ‘DesignSpace‘ class object

Public fields

`weights` A vector denoting the prior weighting of each Design in the design space. Required if robust optimisation is used based on a weighted average variance over the linked designs. If it is not specified in the call to ‘new()‘ then designs are assumed to have equal weighting.

`experimental_condition` A vector indicating the unique identifier of the experimental condition for each observation/row in the matrices X and Z.

Methods

Public methods:

- `DesignSpace$new()`
- `DesignSpace$add()`
- `DesignSpace$remove()`

- DesignSpace$print()
- DesignSpace$n()
- DesignSpace$optimal()
- DesignSpace$show()
- DesignSpace$clone()

Method `new()`**:** Create a new Design Space

Creates a new design space from one or more glmmr designs.

Usage:

```
DesignSpace$new(..., weights = NULL, experimental_condition = NULL)
```

Arguments:

`...` One or more glmmrBase Model objects. The designs must have an equal number of observations.

`weights` Optional. A numeric vector of values between 0 and 1 indicating the prior weights to assign to each of the designs. The weights are required for optimisation, if a weighted average variance is used across the designs. If not specified then designs are assumed to have equal weighting.

`experimental_condition` Optional. A vector of the same length as the number of observations in each design indicating the unique identifier of the experimental condition that observation belongs to, see Details. If not provided, then it is assumed that all observations are separate experimental conditions.

Returns: A 'DesignSpace' object

Examples:

```
\dontshow{
glmmrBase::setParallel(FALSE) # for the CRAN check
setParallelOptim(FALSE)
}
df <- nelder(~ ((int(2)*t(3)) > cl(3)) > ind(5))
df$int <- df$int - 1
des <- Model$new(covariance = list(formula = ~ (1|gr(cl)) + (1|gr(cl,t)),
                                   parameters = c(0.04,0.01)),
                 mean = list(formula = ~ int + factor(t) - 1,
                             parameters = rep(0,4)),
                 data=df,
                 family=gaussian())
ds <- DesignSpace$new(des)
#add another design
des2 <- Model$new(covariance = list(formula = ~ (1|gr(cl)) + (1|gr(cl,t)),
                                   parameters = c(0.05,0.8)),
                 mean = list(formula = ~ int + factor(t) - 1,
                             parameters = rep(0,4)),
                 data=df,
                 family=gaussian())
ds$add(des2)
#report the size of the design
ds$n()
```

```
#we can access specific designs
ds$show(2)$n()
#and then remove it
ds$remove(2)
#or we could add them when we construct object
ds <- DesignSpace$new(des,des2)
#we can specify weights
ds <- DesignSpace$new(des,des2,weights=c(0.1,0.9))
#and add experimental conditions
ds <- DesignSpace$new(des,des2,experimental_condition = df$cl)
```

Method `add()`**:** Add a design to the design space

Usage:

```
DesignSpace$add(x)
```

Arguments:

`x` A 'Design' to add to the design space

Returns: Nothing

Examples:

```
#See examples for constructing the class
```

Method `remove()`**:** Removes a design from the design space

Usage:

```
DesignSpace$remove(index)
```

Arguments:

`index` Index of the design to remove

Returns: Nothing

Examples:

```
#See examples for constructing the class
```

Method `print()`**:** Print method for the Design Space

Usage:

```
DesignSpace$print()
```

Arguments:

`...` ignored

Returns: Prints to the console all the designs in the design space

Examples:

```
#See examples for constructing the class
```

Method `n()`**:** Returns the size of the design space and number of observations

Usage:

```
DesignSpace$n()
```

Examples:

```
#See examples for constructing the class
```

Method `optimal()`**:** Approximate c-optimal design of size m

Algorithms to identify an approximate c-optimal design of size m within the design space.

Usage:

```
DesignSpace$optimal(
  m,
  C,
  attenuate_pars = FALSE,
  V0 = NULL,
  rm_cols = NULL,
  keep = FALSE,
  verbose = TRUE,
  algo = c(1),
  use_combin = FALSE,
  robust_log = FALSE,
  kr = FALSE,
  p,
  tol = 1e-08
)
```

Arguments:

`m` A positive integer specifying the number of experimental conditions to include.

`C` Either a vector or a list of vectors of the same length as the number of designs, see Details.

`attenuate_pars` Logical indicating whether to adapt the marginal expecation in non-linear models

`V0` Optional. If a Bayesian c-optimality problem then this should be a list of prior covariance matrices for the model parameters the same length as the number of designs.

`rm_cols` Optional. A list of vectors indicating columns of X to remove from each design, see Details.

`keep` Logical indicating whether to "keep" the optimal design in the linked design objects and remove any experimental conditions and columns that are not part of the optimal design. Irreversible, so that these observations will be lost from the linked design objects. Defaults to FALSE.

`verbose` Logical indicating whether to reported detailed output on the progress of the algorithm. Default is TRUE.

`algo` A vector of integers indicating the algorithm(s) to use. 1 = local search, 2 = greedy search, 3 = reverse greedy search. Declaring ‘algo = 1‘ for example will use the local search. Providing a vector such as ‘c(3,1)‘ will execute the algorithms in order, so this would run a reverse greedy search followed by a local search. Note that many combinations will be redundant. For example, running a greedy search after a local search will not have any effect.

`use_combin` Logical. If the experimental conditions are uncorrelated, if this option is TRUE then the hill climbing algorithm will be used, otherwise if it is FALSE, then a fast approximate alternative will be used. See Details

`robust_log` Logical. If TRUE and there are multiple designs in the design space then the robust criterion will be a sum of the log of the c-optimality criterion weighted by the study weights, and if FALSE then it will be a weighted sum of the absolute value.

`kr` Logical. Whether to use the Kenwood-Roger small sample bias corrected variance matrix for the fixed effect parameters. We do not recommend using this as it can produce some strange results and its behaviour is not well understood.

`p` Optional. Positive integer specifying the size of the starting design for the greedy algorithm

`tol` Optional scalar specifying the termination tolerance of the Girling algorithm.

Returns: A named list. If using the weighting method then the list contains the optimal experimental weights and a list of exact designs of size 'm', see apportion. If using a combinatorial algorithm then the list contains the rows in the optimal design, the indexes of the experimental conditions in the optimal design, the variance from this design, and the total number of function evaluations. Optionally the linked designs are also modified (see option 'keep').

Examples:

```
\dontshow{
glmmrBase::setParallel(FALSE) # for the CRAN check
setParallelOptim(FALSE)
}
df <- nelder(~(cl(6)*t(5)) > ind(5))
df$int <- 0
df[df$t >= df$cl, 'int'] <- 1
des <- Model$new(
  covariance = list(formula = ~ (1|gr(cl)),
                    parameters = c(0.05)),
  mean = list(formula = ~ factor(t) + int - 1,
              parameters = c(rep(0,5),0.6)),
  data=df,
  family=gaussian(),
  var_par = 1
)
ds <- DesignSpace$new(des)

#find the optimal design of size 30 individuals using reverse greedy search
# change algo=1 for local search, and algo = 2 for greedy search
opt2 <- ds$optimal(30,C=list(c(rep(0,5),1)),algo=3)

#let the experimental condition be the cluster
# these experimental conditions are independent of one another
ds <- DesignSpace$new(des,experimental_condition = df$cl)
# now find the optimal 4 clusters to include
# approximately, finding the weights for each condition
opt <- ds$optimal(4,C=list(c(rep(0,5),1)))
# or use the local search algorithm
opt <- ds$optimal(4,C=list(c(rep(0,5),1)),use_combin = TRUE,algo=1)

#robust optimisation using two designs
des2 <- des$clone(deep=TRUE)
des2$covariance <- Covariance$new(
  data = df,
  formula = ~ (1|gr(cl)*ar1(t)),
  parameters = c(0.25,0.8)
```

```
)
ds <- DesignSpace$new(des,des2)
#weighted average assuming equal weights using local search
\donttest{
opt <- ds$optimal(30,C=list(c(rep(0,5),1),c(rep(0,5),1)),algo=1)
}
```

Method `show()`**:** Returns a linked design

Usage:

```
DesignSpace$show(i)
```

Arguments:

`i` Index of the design to return

Examples:

```
#See examples for constructing the class
```

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
DesignSpace$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `DesignSpace$new`
## ------------------------------------------------


df <- nelder(~ ((int(2)*t(3)) > cl(3)) > ind(5))
df$int <- df$int - 1
des <- Model$new(covariance = list(formula = ~ (1|gr(cl)) + (1|gr(cl,t)),
                                   parameters = c(0.04,0.01)),
                 mean = list(formula = ~ int + factor(t) - 1,
                             parameters = rep(0,4)),
                 data=df,
                 family=gaussian())
ds <- DesignSpace$new(des)
#add another design
des2 <- Model$new(covariance = list(formula = ~ (1|gr(cl)) + (1|gr(cl,t)),
                                   parameters = c(0.05,0.8)),
                 mean = list(formula = ~ int + factor(t) - 1,
                             parameters = rep(0,4)),
                 data=df,
                 family=gaussian())
ds$add(des2)
#report the size of the design
ds$n()
#we can access specific designs
```

```
ds$show(2)$n()
#and then remove it
ds$remove(2)
#or we could add them when we construct object
ds <- DesignSpace$new(des,des2)
#we can specify weights
ds <- DesignSpace$new(des,des2,weights=c(0.1,0.9))
#and add experimental conditions
ds <- DesignSpace$new(des,des2,experimental_condition = df$cl)

## ------------------------------------------------
## Method `DesignSpace$add`
## ------------------------------------------------

#See examples for constructing the class

## ------------------------------------------------
## Method `DesignSpace$remove`
## ------------------------------------------------

#See examples for constructing the class

## ------------------------------------------------
## Method `DesignSpace$print`
## ------------------------------------------------

#See examples for constructing the class

## ------------------------------------------------
## Method `DesignSpace$n`
## ------------------------------------------------

#See examples for constructing the class

## ------------------------------------------------
## Method `DesignSpace$optimal`
## ------------------------------------------------


df <- nelder(~(cl(6)*t(5)) > ind(5))
df$int <- 0
df[df$t >= df$cl, 'int'] <- 1
des <- Model$new(
  covariance = list(formula = ~ (1|gr(cl)),
                    parameters = c(0.05)),
  mean = list(formula = ~ factor(t) + int - 1,
              parameters = c(rep(0,5),0.6)),
  data=df,
  family=gaussian(),
  var_par = 1
)
ds <- DesignSpace$new(des)
```

```
#find the optimal design of size 30 individuals using reverse greedy search
# change algo=1 for local search, and algo = 2 for greedy search
opt2 <- ds$optimal(30,C=list(c(rep(0,5),1)),algo=3)

#let the experimental condition be the cluster
# these experimental conditions are independent of one another
ds <- DesignSpace$new(des,experimental_condition = df$cl)
# now find the optimal 4 clusters to include
# approximately, finding the weights for each condition
opt <- ds$optimal(4,C=list(c(rep(0,5),1)))
# or use the local search algorithm
opt <- ds$optimal(4,C=list(c(rep(0,5),1)),use_combin = TRUE,algo=1)

#robust optimisation using two designs
des2 <- des$clone(deep=TRUE)
des2$covariance <- Covariance$new(
  data = df,
  formula = ~ (1|gr(cl)*ar1(t)),
  parameters = c(0.25,0.8)
)
ds <- DesignSpace$new(des,des2)
#weighted average assuming equal weights using local search

opt <- ds$optimal(30,C=list(c(rep(0,5),1),c(rep(0,5),1)),algo=1)


## ------------------------------------------------
## Method `DesignSpace$show`
## ------------------------------------------------

#See examples for constructing the class
```

`setParallelOptim` *Disable or enable parallelised computing*

Description

By default, the package will use multithreading for many calculations if OpenMP is available on the system. For multi-user systems this may not be desired, so parallel execution can be disabled with this function.

Usage

```
setParallelOptim(parallel_, cores_ = 2L)
```

Arguments

`parallel_`	Logical indicating whether to use parallel computation (TRUE) or disable it (FALSE)
`cores_`	Number of cores for parallel execution

Value

None, called for effects

Index

apportion, 2, *4*, *8*

DesignSpace, 3

Model, *3*, *5*

setParallelOptim, 11