Package ‘rsm’

October 14, 2022

Version 2.10.3

Date 2021-10-06

Title Response-Surface Analysis

Description Provides functions to generate response-surface designs,
fit first- and second-order response-surface models,
make surface plots, obtain the path of steepest ascent,
and do canonical analysis. A good reference on these methods
is Chapter 10 of Wu, C-F J and Hamada, M (2009)
``Experiments: Planning, Analysis, and Parameter Design Optimization''
ISBN 978-0-471-69946-0. An early version of the package is
documented in Journal of Statistical Software <doi:10.18637/jss.v032.i07>.

Suggests emmeans (> 1.3.5.1), Vdgraph, conf.design, DoE.base, FrF2

Imports estimability

License GPL (>= 2)

LazyLoad yes

NeedsCompilation no

Author Russell Lenth [aut, cre]

Maintainer Russell Lenth <russell-lenth@uiowa.edu>

Repository CRAN

Date/Publication 2021-10-06 20:20:02 UTC

R topics documented:

`rsm-package` *Response-surface analysis*

Description

The `rsm` package provides functions useful for designing and analyzing experiments that are done sequentially in hopes of optimizing a response surface.

The function `ccd` can generate (and randomize) a central-composite design; it allows the user to specify an aliasing or fractional blocking structure. The function `bbd` generates and randomizes a Box-Behnken design. The function `ccd.pick` is useful for identifying good parameter choices in central-composite designs. Functions `cube`, `star`, `foldover`, `dupe`, and `djoin` are also provided to build-up designs from individual blocks. The function `varfcn` allows the experimenter to examine the predictive capabilities of a design before collecting data.

The function `rsm` is an enhancement of `lm` that provides for additional analyses peculiar to response surfaces. It requires a model formula that contains a call to `FO` or `SO` to specify a first- or second-order model. Once the model is fitted, the `steepest` function may be used to obtain the direction of steepest ascent (or descent). `canonical.path` is an alternative to `steepest` for second-order response surfaces.

In RSM methods, appropriate coding of data is important not only for numerical stability, but for proper scaling of results; the function `coded.data` and its relatives facilitate this coding requirement.

Finally, a few more functions are provided that may be useful beyond response-surface applications. `contour.lm`, `persp.lm`, and `image.lm` aids in visualizing a response surface, or of any other `lm` object where a surface is fitted. `model.data` recovers the data used in a `lm` call, but unlike `model.frame`, no polynomials, factors, etc. are expanded.

For more information and examples, use ‘`vignette("rsm")`’ and ‘`vignette("rs-illus")`’. Additionally, ‘`vignette("rsm-plots")`’ provides some illustrations of the graphics functions.

Author(s)

Russell V. Lenth

Maintainer: Russell V. Lenth <russell-lenth@uiowa.edu>

References

Box, GEP, Hunter, JS, and Hunter, WG (2005) *Statistics for Experimenters* (2nd ed.), Wiley-Interscience.

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

Myers, RH, Montgomery, DC, and Anderson-Cook, CM (2009), *Response Surface Methodology* (3rd ed.), Wiley.

`bbd` *Generate a Box-Behnken design*

Description

This function can generate a Box-Behnken design in 3 to 7 factors, and optionally will block it orthogonally if there are 4 or 5 factors. It can also randomize the design.

Usage

```
bbd(k, n0 = 4, block = (k == 4 | k == 5), randomize = TRUE, coding)
```

Arguments

`k`	A formula, or an integer giving the number of variables. If the formula has a left-hand side, the variables named there are appended to the design and initialized to `NA`.
`n0`	Number of center points in each block.
`block`	Logical value specifying whether or not to block the design; or a character string (taken as `TRUE`) giving the desired name for the blocking factor. Only BBDs with 4 or 5 factors can be blocked. A 4-factor BBD has three orthogonal blocks, and a 5-factor BBD has two.
`randomize`	Logical value determining whether or not to randomize the design. If `block` is `TRUE`, each block is randomized separately.
`coding`	Optional list of formulas. If this is provided, it overrides the default coding formulas.

Details

Box-Behnken designs (BBDs) are useful designs for fitting second-order response-surface models. They use only three levels of each factor (compared with 5 for central-composite designs) and sometimes fewer runs are required than a CCD. This function uses an internal table of BBDs; it only works for 3 to 7 factors.

If `k` is specified as a formula, the names in the formula determine the names of the factors in the generated design. Otherwise, the names will be `x1, x2, ...`. If `coding` is not specified, default codings are created in the form ‘`x ~ x.as.is`’.

Value

A coded.data object with the generated design and the additional valiables run.order and std.order. The blocking variable, if present, will be a factor; all other variables will be numeric.

Note

To avoid aliasing the pure-quadratic terms, you must use a positive value of n0.

The non-exported function rsm:::.bbd.1.41 is provided in case it is needed by other packages for compatibility with old versions of **rsm** (version 1.41 or earlier). Given the same seed, it will also reproduce the randomization as a previously generated design from an old version.

Author(s)

Russell V. Lenth

References

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

Myers, RH, Montgomery, DC, and Anderson-Cook, CM (2009) *Response Surface Methodology* (3rd ed.), Wiley.

See Also

ccd, coded.data

Examples

```
library(rsm)

### Simple 3-factor case, not randomized so structure is evident
bbd(3, randomize=FALSE)

### 5-factor BBD, divided between two plants
bbd(y1 + y2 ~ A + B + C + D + E,  n0 = 5,  block = "Plant")
```

---

ccd *Generate central-composite designs and associated building blocks*

Description

These functions generate central-composite designs, or building blocks thereof. They allow for flexible choices of replications, aliasing of predictors and fractional blocks, and choices of axis or ‘star’ points.

Usage

```
cube(basis, generators, n0 = 4, reps = 1, coding, randomize = TRUE,
    blockgen, bid = 1, inscribed = FALSE)
star(basis, n0 = 4, alpha = "orthogonal", reps = 1, randomize = TRUE)
dupe(design, randomize = TRUE, coding)
foldover(basis, variables, bid, randomize = TRUE)
ccd(basis, generators, blocks = "Block", n0 = 4, alpha = "orthogonal",
    wbreps = 1, bbreps = 1, randomize = TRUE, inscribed = FALSE,
    coding, oneblock = FALSE)
```

Arguments

`basis`	In `cube` and `ccd`, a formula, or an integer giving the number of variables. If the formula has a left-hand side, the variables named there are appended to the design and initialized to `NA`. In `star`, `dupe`, and `foldover`, `basis` is a `coded.data` object to use as a reference.
`generators`	Optional formula or list of formulas to generate aliased variables
`n0`	Integer giving the number of center points. In `ccd`, this can be a vector of two numbers for the numbers of center points in the cube blocks and the star blocks, respectively.
`reps`	Integer number of replications of the cube or the star. (This does *not* create replicate blocks; use `djoin` to do that.)
`coding`	List of coding formulas for the design variables (those in `basis` and `generators`). In `dupe`, `coding` may be used to change the coding formulas, e.g. in a situation where we want to use the same design as before but center it elsewhere.
`randomize`	Logical value determining whether or not to randomize the design. In `ccd`, each block is randomized separately.
`blockgen`	A formula, string, or list thereof. Each element is evaluated, and the distinct combinations define fractional blocks for the design. Unlike `ccd`, `cube` returns only one of these blocks.
`bid`	(For block ID.) An integer index (from 1 to number of blocks) of the fractional block to return. The indexes are defined by the standard ordering of the block generators; e.g. if `blockgen` is of length 2, the `bid` values of (1, 2, 3, 4) correspond to generated levels of (`--`, `+-`, `-+`, `++`) respectively.
`inscribed`	Logical value; if `FALSE`, the cube points are at +/- 1 in each variable. If `TRUE`, the entire desgn is scaled down so that the axis points are at +/- 1 and the cube points are at interior positions. In `cube` only, `inscribed` may be given a numeric value: use the value of `alpha` anticipated when axis points are added; or use '`inscribed = TRUE`' to scale in anticipation of '`alpha = "spherical"`'.
`alpha`	If numeric, the position of the 'star' points. May also be a character string that matches or partially matches one of these: `"orthogonal"` the star points are positioned to block the design orthogonally `"rotatable"` the star points are chosen to make the design rotatable `"spherical"` the star points are the same distance as the corners of the design cube (alpha is the square root of the number of design factors)

	`"faces"` the star points are face-centered (same as ‘`alpha = 1`’)
	The user may specify a vector value of `alpha` if it is desired to vary them on different axes. The values are rotated cyclically as needed.
`design`	A `coded.data` object to be duplicated.
`blocks`	A string or a formula. If a character string, it is the name of the blocking factor; if a formula, the left-hand side is used as the name of the blocking factor. The formula(s) on the right-hand side are used to generate separate fractional blocks.
`variables`	Character vector of names of variables to fold over.
`wbreps`	Number(s) of within-block replications. If this is a vector of length 2, then separate numbers are used for the ‘cube’ and the ‘star’ blocks respectively.
`bbreps`	Number(s) of between-block replications (i.e., number of repeats of each block). If this is a vector of length 2, then separate numbers are used for the ‘cube’ and the ‘star’ blocks respectively.
`oneblock`	Logical. If `TRUE`, the blocking factor is removed and the whole design is randomized as a single block. Note that the default number of center points may be larger than you anticipated because they are combined.

Details

Central-composite designs (CCDs) are popular designs for use in response-surface exploration. They are blocked designs consisting of at least one ‘cube’ block (two-level factorial or fractional factorial, plus center points), and at least one ‘star’ block (points along each axis at positions `-alpha` and `+alpha`), plus center points. Everything is put on a coded scale, where the cube portion of the design has values of -1 and 1 for each variable, and the center points are 0.

The `ccd` function creates an entire CCD design; however, in practice, we often start with just the cube portion and build from there. Therefore, the functions `cube`, `star`, `dupe`, and `foldover` are provided, and one may use `djoin` to combine them.

In `cube` and `ccd`, the `basis` argument determines a basic design used to create cube blocks. For example, ‘`cube(basis = ~ A + B + C)`’ would generate a basic design of 8 factorial points plus center points. Use `generators` if you want additional variables in a fractional design; for example, ‘`generators = c(D ~ -A*B, E ~ B*C)`’ added to the above would generate a 5-factor design with defining relation `I = -ABD = BCE = -ACDE`. For convenience, `basis` may be an integer instead of a formula, in which case default variable names of `x1, x2, ...` are used; for example, ‘`cube(3, ~ -x1*x2*x3)`’ generates a 1/2 fraction design with added center points.

If you want the cube points divided into fractional blocks, give the formula(s) in the `blockgen` argument of `cube`, or the `blocks` argument of `ccd`. For instance, suppose we call ‘`cube(basis = A+B+C+D+E`’, ‘`generators = F~-A*C*D)`’. This design has 32 runs; but adding the argument ‘`blockgen = c("A*B*C","C*D*E")`’ will create a fractional block of 32/4 = 8 runs. (`cube` is flexible; we could have used a formula instead, either ‘`blockgen = ~ c(A*B*C, C*D*E)`’ or ‘`blockgen = c(~A*B*C, ~C*D*E)`’.) Center points are added to each block as specified. In a call to `ccd` with the same `basis` and `generators`, adding ‘`blocks = Day ~ c(A*B*C, C*D*E)`’ would do the same thing, only all 4 blocks will be included, and a factor named `Day` distinguishes the blocks.

The functions `star`, `dupe`, and `foldover` provide for creating new design blocks based on an existing design. They also provide for delayed evaluation: if the `basis` argument is missing, these functions simply return the call, `djoin` will fill-in ‘`basis = design1`’ and evaluate it.

`dupe` simply makes a copy of the design, and re-randomizes it. Therefore it is also a convenient way to re-randomize a design. If `coding` is provided, the coding formulas are replaced as well – for example, to re-center the design.

Use `star` to generate star (axis) points, which consist of center points plus points at `+/- alpha` on each coordinate axis. You may specify the `alpha` you want, or a character argument to specify a certain criterion be met. For example, using delayed evaluation, ‘`ccd1 = djoin(cube1, star(alpha="sph"))`’ will return a CCD with `cube1` as the cube block, and with axis points at the same distance as the corners of the cube. Conditions for the criteria on `alpha` are described in detail in references such as Myers *et al.* (2009).

In `star`, determinations of orthogonality and rotatability are based on computed design moments of `basis`, rather than any assumptions about the structure of the design being augmented. Thus, it may be possible to augment an unusual design to obtain a rotatable design. Also, if an orthogonal star block is requested, the value of `alpha` may vary from axis to axis if that is required to satisfy the condition.

`foldover` reverses the levels of one or more design variables (i.e., those that are coded). By default, it reverses them all. However, if the `bid` argument is supplied, it instead returns the `bid`th fractional block that `cube` would have generated. That is, ‘`foldover(des, bid=3)`’ is equivalent to ‘`cube(<arguments that created des>, bid=3)`’ – only it does so much more efficiently by folding on the appropriate factors.

In cases where there are constraints on the region of operability, you may want to specify `inscribed = TRUE`. This will scale-down the design so that no coded value exceeds 1. If using a building-block approach starting with a first-order design from `cube`, call `cube` with `inscribed` set to the anticipated value of `alpha`, or use ‘`inscribed = TRUE`’, and then use ‘`alpha = "spherical"`’ in the subsequent call to `star`.

`ccd` generates an entire CCD. In practice, the building-block approach with `cube`, `star`, etc. is usually preferable, but `ccd` exists for convenience and backward compatibility with pre-2.00 versions of **rsm**. Many of the arguments are the same as those in `cube`; however, `n0`, `wbreps`, `bbreps` may be single values or vectors; if vectors, the first element is for the cube portions and the second element is for the star portions. In `ccd`, specifying `wbreps` is equivalent to specifying `reps` in a call to `cube` or `star`. `bbreps` refers to replicate blocks in the experiment, so that ‘`bbreps = c(2,3)`’ specifies that we join two cube blocks and three blocks of star points.

If `coding` is not specified in a new design, default identity codings are created, e.g. ‘`x1 ~ x1.as.is`’.

Value

A `coded.data` object with the generated design, with additional variables `run.order` and `std.order`. If a multi-block design, the generated blocking variable will be a `factor`; all other variables will be numeric. The designs are sorted by blocks and `run.order` within blocks; and (unlike pre-1.41 versions of **rsm**) the `row.names` will be integers corresponding to this ordering. The user may sort by block and `std.order` within block to display the designs in their pre-randomized order.

Note

Poor choices of `generators` and/or `blocks` can alias or partially alias some effects needed to estimate a second-order response surface. It is a good idea to run `varfcn` before collecting data to examine the prediction capabilities of the design and to ensure that the desired model can be fitted.

The function `ccd.pick` is available to help determine good choices for arguments to `cube`, `star`, and `ccd`.

An alternative to a CCD when you want to go straight to second-order modeling is a Box-Behnken design, generated by `bbd`. These designs are not as various or flexible as CCDs, but they can require fewer runs.

The non-exported function `rsm:::.ccd.1.41` is provided in case it is needed by other packages for compatibility with old versions of **rsm** (version 1.41 or earlier). Given the same seed, it will also reproduce the randomization as a previously generated design from an old version.

Author(s)

Russell V. Lenth

References

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

Myers, RH, Montgomery, DC, and Anderson-Cook, CM (2009) *Response Surface Methodology* (3rd ed.), Wiley.

See Also

`ccd.pick`, `coded.data`, `varfcn`, `bbd`

Examples

```
library(rsm)

### Generate a standard 3-variable first-order design with 8 corner points and 4 center points
( FOdes <- cube (3, n0 = 4, coding = list (
                x1 ~ (Temp - 150)/10, x2 ~ (Pres - 50)/5, x3 ~ Feedrate - 4)) )

### Add an orthodonal star block with 12 runs to create a second-order CCD
( SOdes <- djoin(FOdes, star(n0=6)) )

### Same as above, except make the whole CCD at once; and make it rotatable
### and inscribed so that no coded value exceeds 1
SOdes2 <- ccd (3, n0 = c(4,6), alpha = "rotatable", inscribed = TRUE, coding = list (
                x1 ~ (Temp - 150)/10, x2 ~ (Pres - 50)/5, x3 ~ Feedrate - 4))

### Make two replicate blocks of FOdes (2nd one randomized differently)
djoin(FOdes, dupe(FOdes))

### Fractional blocking illustration (with no center points)
# Basic design (bid = 1 ---> block generators b1 = -1, b2 = -1)
block1 <- cube (~ x1 + x2 + x3 + x4,  generators = x5 ~ x1 * x2 * x3 * x4,
                n0 = 0, blockgen = ~ c(x1 * x2, x1 * x3), bid = 1)
block1

# The foldover (on all variables) of block1, in the same order
```

```
foldover(block1, randomize=FALSE)

# The 4th fractional block:
( block4 <- foldover(block1, bid = 4) )
```

---

ccd.pick *Find a good central-composite design*

Description

This function looks at all combinations of specified design parameters for central-composite designs, calculates other quantities such as the `alpha` values for rotatability and orthogonal blocking, imposes specified restrictions, and outputs the best combinations in a specified order. This serves as an aid in identifying good designs. The design itself can then be generated using `ccd`, or in pieces using `cube`, `star`, etc.

Usage

```
ccd.pick(k, n.c = 2^k, n0.c = 1:10, blks.c = 1, n0.s = 1:10, bbr.c = 1,
         wbr.s = 1, bbr.s = 1, best = 10, sortby = c("agreement", "N"),
         restrict)
```

Arguments

`k`	Number of factors in the design
`n.c`	Number(s) of factorial points in each cube block
`n0.c`	Numbers(s) of center points in each cube block
`blks.c`	Number(s) of cube blocks that together comprise one rep of the cube portion
`n0.s`	Numbers(s) of center points in each star (axis-point) block
`bbr.c`	Number(s) of copies of each cube block
`wbr.s`	Number(s) of replications of each star poit within a block
`bbr.s`	Number(s) of copies of each star block
`best`	How many designs to list. Use `best=NULL` to list them all
`sortby`	String(s) containing numeric expressions that are each evaluated and used as sorting key(s). Specify `sortby=NULL` if no sorting is desired.
`restrict`	Optional string(s) containing Boolean expressions that are each evaluated. Only combinations where all expressions are `TRUE` are retained.

Details

A grid is created with all combinations of `n.c`, `n0.c`, ..., `bbr.s`. Then for each row of the grid, several additional variables are computed:

`n.s` The total number of axis points in each star block

`N` The total number of observations in the design

`alpha.rot` The position of axis points that make the design rotatable. Rotatability is achieved when design moment [iiii] = 3[iijj] for i and j unequal.

`alpha.orth` The position of axis points that make the blocks mutually orthogonal. This is achieved when design moments [ii] within each block are proprtional to the number of observations within the block.

`agreement` The absolute value of the log of the ratio of `alpha.rot` and `alpha.orth`. This measures agreement between the two `alpha`s.

If `restrict` is provided, only the cases where the expressions are all `TRUE` are kept. (Regardless of `restrict`, rows are eliminated where there are insufficient degrees of freedom to estimate all needed effects for a second-order model.) The rows are sorted according to the expressions in `sortby`; the default is to sort by `agreement` and `N`, which is suitable for finding designs that are both rotatable and orthogonally blocked.

Value

A `data.frame` containing `best` or fewer rows, and variables `n.c`, `n0.c`, `blks.c`, `n.s`, `n0.s`, `bbr.c`, `wbr.s`, `bbr.s`, `N`, `alpha.rot`, and `alpha.orth`, as described above.

Author(s)

Russell V. Lenth

References

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

Myers, RH, Montgomery, DC, and Anderson-Cook, CM (2009) *Response Surface Methodology* (3rd ed.), Wiley.

See Also

`ccd`

Examples

```
library(rsm)

### List CCDs in 3 factors with between 10 and 14 runs per block
ccd.pick(3, n0.c=2:6, n0.s=2:8)
# (Generate the design that is listed first:)
# ccd(3, n0=c(6,4))
```

```
### Find designs in 5 factors containing 1, 2, or 4 cube blocks
### of 8 or 16 runs, 1 or 2 reps of each axis point,
### and no more than 70 runs altogether
ccd.pick(5, n.c=c(8,16), blks.c=c(1,2,4), wbr.s=1:2, restrict="N<=70")
```

ChemReact *Chemical Reaction Data*

Description

These data are from a central composite design with 2 factors in 2 blocks. The design variables are in actual, not coded, form.

Usage

```
ChemReact
ChemReact1
ChemReact2
```

Format

A data frame with 14 observations on the following 4 variables.

`Time` a numeric vector; design variable with settings of 80, 85, and 90.

`Temp` a numeric vector; design variable with settings of 170, 175, and 180.

`Block` a factor with levels `B1` `B2`. Block `B1` is a first-order design with 3 center points. Block `B2` consists of axis points and 3 more center points.

`Yield` a numeric vector; response variable: yield of the chemical process.

`ChemReact1` and `ChemReact2` are the separate blocks. Each has 7 runs and three variables (`Block` is excluded from these).

Source

Table 7.6 of Myers, RH, Montgomery, DC, and Anderson-Cook, CM (2009), *Response Surface Methodology* (3rd ed.), Wiley.

codata *Automobile emissions data*

Description

This is a replicated 3^2 experiment reported in Box, Hunter, and Hunter (2005), Table 10.17.

Usage

```
codata
```

Format

A data frame with 18 observations on the following 3 variables.

`x1` a numeric vector, coded design variable for ethanol concentration

`x2` a numeric vector, coded design variable for air-to-fuel ratio

`y` a numeric vector, the response (CO concentration, in micrograms per cubic meter)

Details

This example, when fitted with a second-order response surface, is an example of a rising ridge. The dataset is duscussed again one chapter later in the source text; Figure 11.17 of BH^2 suggests the coding formulas used in the example below.

Source

Box, GEP, Hunter, JS, and Hunter, WG (2005) *Statistics for Experimenters* (2nd ed), Wiley.

Examples

```
# Create a coded dataset based on info in BH^2 Fig 11.17
CO <- as.coded.data(codata,  x1 ~ (Ethanol - 0.2)/0.1,  x2 ~ A.F.ratio - 15)
names(CO)[3] <- "CO.conc"
```

coded.data *Functions for coded data*

Description

These functions facilitate the use of coded data in response-surface analysis.

Usage

```
coded.data(data, ..., formulas = list(...), block = "block")
as.coded.data(data, ..., formulas = list(...), block = "block")
decode.data(data)
recode.data(data, ..., formulas = list(...))

val2code(X, codings)
code2val(X, codings)

## S3 method for class 'coded.data'
print(x, ..., decode = TRUE)

### --- Methods for managing coded data ---
is.coded.data(x)

## S3 method for class 'coded.data'
x[...]

codings(object)
## S3 method for class 'coded.data'
codings(object)
codings(object) <- value

## S3 replacement method for class 'coded.data'
names(x) <- value

## Generic method for true variable names (i.e. decoded names)
truenames(x)
## S3 method for class 'coded.data'
truenames(x)
## Generic replacement method for truenames
truenames(x) <- value
## S3 replacement method for class 'coded.data'
truenames(x) <- value
```

Arguments

data	A data.frame
formulas	List of coding formulas; see details
block	Name(s) of blocking variable(s). It is pmatched (case insensitively) with names in data to identify blocking factorss
X	A vector, matrix, or data.frame to be coded or decoded.
codings	A list of formulas; see Details
decode	Logical. If TRUE, the decoded values are displayed; if FALSE, the codings are displayed.
object	A coded.data object

`x`	A `coded.data` object
`value`	Replacement value for `<-` methods
`...`	In `coded.data`, `as.coded.data`, and `recode.data`, `...` allows specifying formulas as arguments rather than as a list. In other functions, `...` is passed to the parent methods.

Details

Typically, coding formulas are of the form `x ~ (var - center) / halfwd` where `x` and `var` are variable names, and `center` and `halfwd` are numbers. The left-hand side gives the name of the coded variable, and the right-hand side should be a linear expression in the uncoded variable (linearity is *not* explicitly checked, but nonlinear expressions will not decode correctly.) If `coded.data` is called without formulas, automatic codings are created (along with a warning message). Automatic codings are based on transforming all non-block variables having five or fewer unique values to the interval [-1,1]. If no formulas are provided in `as.coded.data`, default coding formulas like those for `cube` are created all numeric variables with mean zero – again with a warning message.

An S3 `print` method is provided for the `coded.data` class; it displays the data.frame in either coded or decoded form, along with the coding formulas. Some users may prefer `print.data.frame` or `as.data.frame` in lieu of `print` with '`decode=FALSE`'; they produce the same output without displaying the coding formulas.

Use `coded.data` to convert a `data.frame` in which the variables are on their original scales. The variables named in the formulas are coded and replaced with their coded versions (and also renamed).

In contrast, `as.coded.data` does not modify any of the data; it assumes the variables are already coded, and the coding information is simply added. In addition, if `data` is already a `coded.data` object from a pre-1.41 version of **rsm**, it is converted to be compatible with new capabilities such as `djoin` (no `formulas` argument is needed in this case). Any blocking factors should be specified in the `blocks` argument.

`decode.data` converts a dataset of class `coded.data` and returns a `data.frame` containing the original variables.

`recode.data` is used to convert a `coded.data` object to new codings. Important: this *changes* the coded values to match the new coding formulas. If you want to keep the coded values the same, but change the levels they represent, use '`codings(object) <- \dots`' or `dupe`.

`code2val` converts coded values to the original scale using the codings provided, and returns an object of the same class as `X`. `val2code` converts the other direction. When using these functions, it is essential that the names (or column names in the case of matrices) match those of the corresponding coded or uncoded variables.

`codings` is a generic function for accessing codings. It returns the list of coding formulas from a `coded.data` object. One may use an expression like '`codings(object) <- list(\dots)`' to change the codings (without changing the coded values themselves). See also `codings.rsm`.

`is.coded.data(x)` returns `TRUE` if `x` inherits from `coded.data`, and `FALSE otherwise.`

The extraction function `x[...]` and the naming functions `names<-`, `truenames`, and `truenames<-` are provided to preserve the integrity of codings. For example, if `x[, 1:3]` excludes any coded columns, their coding formulas are also excluded. If all coded columns are excluded, the return value is unclassed from `coded.data`. When variable names are changed using `names(x) <- ...`, the coding formulas are updated accordingly. The `truenames` function returns the names of the variables in the decoded dataset. We can change the decoded names using `truenames(x) <- ...`, and the coding formulas are updated. Note that `truenames` and `truenames<-` work the same as `names` and `names<-` for unencoded variables in the object.

Another convenient way to copy and change the coding formulas a coded dataset (and optionally re-randomize it) is to use the `dupe` function with a `coding` argument.

When a design is created in another package, some of the variables may be `factors`, in which case they are converted using `as.numeric` (values of 1, 2, ...). These levels may be regarded as a yet different coding of the variables, and so it may take two steps to get it in the desired form: one to convert the supplied levels to the desired range (often -1 to 1), and the other to replace the coding formulas to correspond to the real values of the variables to be used. See the examples.

Value

`coded.data`, `as.coded.data`, and `recode.data` return an object of class `coded.data`, which inherits from `data.frame`. A `coded.data` object is stored in coded form, and its `names` attribute contains the coded names, where they apply. Thus, when fitting models in `rsm` or `lm` with coded data as the `data` argument, the model formula should be given in terms of the coded variables.

Note

Starting with **rsm** version 2.00, the `coded.data` class involves additional attributes to serve broader needs in design-generation. Because of this, old `coded.data` objects may need to be updated using `as.coded.data` if they are to be used with the newer functions such as `djoin`.

Author(s)

Russell V. Lenth

References

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

See Also

`data.frame`, `djoin`, `dupe`, `rsm`

Examples

```
library(rsm)

### Existing dataset with variables on actual scale
CR <- coded.data (ChemReact, x1 ~ (Time - 85)/5, x2 ~ (Temp - 175)/5)
CR                              # same as print(CR, decode = TRUE)
print(CR, decode = FALSE)       # similar to as.data.frame(CR)
code2val (c(x1=.5, x2=-1), codings = codings(CR))

### Existing dataset, already in coded form
CO <- as.coded.data(codata, x1 ~ (Ethanol - 0.2)/0.1, x2 ~ A.F.ratio - 15)
```

```
truenames(CO)
names(CO)

# revert x2 to an uncoded variable
codings(CO)[2] <- NULL
truenames(CO)

### Import a design that is coded in a different way

if (require(conf.design)) { # ----- This example requires conf.design -----

# First, generate a 3^3 in blocks and import it via coded.data
    des3 <- coded.data(conf.design(p=3, G=c(1,1,2)))
    # NOTE: This returns a warning message but does the right thing --
    # It generates these names and coding formulas automatically:
    #   x1 ~ (T1 - 2)/1
    #   x2 ~ (T2 - 2)/1
    #   x3 ~ (T3 - 2)/1
# Now randomize and change the codings and variable names for the real situation:
    mydes <- dupe(des3, coding = c(x1 ~ (Dose - 20)/5,  x2 ~ (Conc - 40)/10,
                             x3 ~ (Time - 60)/15))

} # ----- end of example requiring package conf.design -----
```

contour.lm *Surface plot(s) of a fitted linear model*

Description

contour, image, and persp methods that display the fitted surface for an lm object involving two or more numerical predictors.

Usage

```
## S3 method for class 'lm'
contour(x, form, at, bounds, zlim, xlabs, hook,
    plot.it = TRUE, atpos = 1, decode = TRUE, image = FALSE,
    img.col = terrain.colors(50), ...)

## S3 method for class 'lm'
image(x, form, at, bounds, zlim, xlabs, hook,
    atpos = 1, decode = TRUE, ...)

## S3 method for class 'lm'
persp(x, form, at, bounds, zlim, zlab, xlabs,
    col = "white", contours = NULL, hook, atpos = 3, decode = TRUE,
    theta = -25, phi = 20, r = 4, border = NULL, box = TRUE,
    ticktype = "detailed", ...)
```

Arguments

`x`	A `lm` object.
`form`	A formula, or a list of formulas.
`at`	Optional *named* list of fixed values to use for surface slices. For example, if the predictor variables are `x1`, `x2`, and `x3`, the contour plot of `x2` versus `x1` would be based on the fitted surface sliced at the `x3` value specified in `at`; the contour plot of `x3` versus `x1` would be sliced at the `at` value for `x2`; etc. If not provided, `at` defaults to the mean value of each numeric variable.
`bounds`	Optional *named* list of bounds or grid values to use for the variables having the same names. See details.
`zlim`	`zlim` setting passed to parent methods `contour`, `image`, or `persp`. The same `zlim` is used in all plots when several are produced. If not provided, the range of values across all plotted surfaces is used.
`zlab`	Optional label for the vertical axis.
`xlabs`	Alternate labels for predictor axes (see Details).
`hook`	Optional list that can contain functions `pre.plot` and `post.plot`. May be used to add annotations or to re-route the graphs to separate files (see Details).
`atpos`	Determines where `at` values are displayed. A value of 1 (or 2) displays it as part of the *x* (or *y*) axis label. A value of 3 displays it as a subtitle below the plot. A value of 0 suppresses it. Any other nonzero value will cause the label to be generated but not displayed; it can be accessed via a `hook` function.
`decode`	This has an effect only if `x` is an `rsm` object or other model object that supports `coded.data`. In such cases, if `decode` is `TRUE`, the coordinate axes are transformed to their decoded values.
`image`	Set to `TRUE` if you want an image plot overlaid by contours.
`img.col`	Color map to use when `image=TRUE`.
`plot.it`	If `TRUE`, no plot is produced, just the return value.
`col`	Color or colors used for facets in the perspective plot (see details).
`contours`	If non-`NULL`, specifications for added contour lines in perspective plot.
`theta, phi`	Viewing angles passed to `persp` (different defaults).
`r`	Viewing distance passed to `persp` (different default).
`border, box`	Options passed to `persp`.
`ticktype`	Option passed to `persp` (different default).
`...`	Additional arguments passed to `contour`, `image`, or `persp`. Note, however, that a `ylab` is ignored, with a message to Use `xlabs` instead.

Details

`form` may be a single formula or a list of formulas. A simple formula like `x2 ~ x1` will produce a contour plot of the fitted regression surface for combinations of `x2` (vertical axis) and `x1` (horizontal axis). A list of several such simple formulas will produce a contour plot for each formula. A two-sided formula produces contour plots for each left-hand variable versus each right-hand variable (except when they are the same); for example, `x1+x3 ~ x2+x3` is equivalent to `list(x1~x2, x3~x2, x1~x3)`. A one-sided formula produces contour plots for each pair of variables. For example, `~ x1+x2+x3` is equivalent to `list(x2~x1, x3~x1, x3~x2)`.

For any variables not in the `bounds` argument, a grid of 26 equally-spaced values in the observed range of that variable is used. If you specify a vector of length 2, it is interpreted as the desired range for that variable and a grid of 26 equally-spaced points is generated. If it is a vector of length 3, the first two elements are used as the range, and the third as the number of grid points. If it is a vector of length 4 or more, those values are used directly as the grid values.

The results are based on the predicted values of the linear model over the specified grid. If there are `factors` among the predictors, the predictions are made over all levels (or combinations of levels) of those factors, and then averaged together. (However, the user may include factors in `at` to restrict this behavior.)

By default, the predictor axes are labeled using the variable names in `form`, unless `x` is an `rsm` or other object that supports `coded.data`, in which case either the decoded variable names or the variable-coding formulas are used to generate axis labels, depending on whether `decode` is `TRUE` or `FALSE`. These axis labels are replaced by the entries in `xlabs` if provided. One must be careful using this to make sure that the names are mapped correctly. The entries in `xlabs` should match the respective unique variable names in `form`, *after sorting them in (case-insensitive) alphabetical order* (not necessarily in order of appearance). Note that if `form` is changed, it may also be necessary to change `xlabs`.

Please note that with models fitted to coded data, coded values should be used in `at` or `bounds`, regardless of whether `decode` is `TRUE` or `FALSE`. However, any elements that are added afterward via `points`, `lines`, etc., must be specified in terms of whatever coordinate system is present in the plots.

In `persp`, contour lines may be added via the `contours` argument. It may be a boolean or character value, or a `list`. If boolean and `TRUE`, default black contour lines are added to the bottom surface of the box. Character values of `"top"`, `"bottom"` add black contour lines to the specified surface of the box. `contours = "colors"` puts contour lines on the bottom using the same colors as those at the same height on the surface. Other character values of `contours` are taken to be the desired color of the contour lines, plotted at the bottom. If `contours` is a named `list`, its elements (all are optional) are used as follows:

`z` Height where the contour lines are plotted. May be `"bottom"` (default), `"top"`, or a numeric value.

`col` Color of the lines. If not specified, they will be black. May be integer color values, color names, or `"colors"` to match the surface colors.

`lwd` Line width; default is 1.

Since these functions often produce several plots, the `hook` argument is provided if special setups or annotations are needed for each plot. It should be a list that defines one or both of the functions `pre.plot` and `post.plot`. Both of these functions have one argument, the character vector `labs` for that plot (see Value documentation).

Additional examples and discussion of these plotting functions is available via `vignette("rsm-plots")`.

Value

A `list` containing information that is plotted. Each list item is itself a `list` with the following components:

`x, y`	The values used for the x and y axes
`z`	The matrix of fitted response values
`labs`	Character vector of length 5: Elements 1 and 2 are the x and y axis labels, elements 3 and 4 are their original variable names, and element 5 is the slice label (empty if `atpos` is 0)
`zlim`	The computed or provided `zlim` values
`transf`	(`persp` only) The 3D transformation for `trans3d`

Author(s)

Russell V. Lenth

References

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

See Also

`contour`

Examples

```
### Basic example with a linear model:
mpg.lm <- lm(mpg ~ poly(hp, disp, degree = 3), data = mtcars)
contour(mpg.lm, hp ~ disp, image = TRUE)

### Extended example with an rsm model...
heli.rsm <- rsm (ave ~ block + SO(x1, x2, x3, x4), data = heli)

# Plain contour plots
par (mfrow = c(2,3))
contour (heli.rsm, ~x1+x2+x3+x4, at = xs(heli.rsm))

# Same but with image overlay, slices at origin and block 2,
# and no slice labeling
contour (heli.rsm, ~x1+x2+x3+x4, at = list(block="2"),
    atpos = 0, image = TRUE)

# Default perspective views
persp (heli.rsm, ~x1+x2+x3+x4, at = xs(heli.rsm))

# Same plots, souped-up with facet coloring and axis labeling
persp (heli.rsm, ~x1+x2+x3+x4, at = xs(heli.rsm),
    contours = "col", col = rainbow(40), zlab = "Flight time",
  xlabs = c("Wing area", "Wing length", "Body width", "Body length"))

## Not run:
### Hints for creating graphics files for use in publications...
```

```
# Save perspective plots in one PDF file (will be six pages long)
pdf(file = "heli-plots.pdf")
persp (heli.rsm, ~x1+x2+x3+x4, at = xs(heli.rsm))
dev.off()

# Save perspective plots in six separate PNG files
png.hook = list(
    pre.plot = function(lab)
        png(file = paste(lab[3], lab[4], ".png", sep = "")),
    post.plot = function(lab)
        dev.off())
persp (heli.rsm, ~x1+x2+x3+x4, at = xs(heli.rsm), hook = png.hook)

## End(Not run)
```

---

djoin *Join designs together into a blocked design*

Description

This implements the **rsm** package's building-block provisions for handling sequences of experiments. We often want to join two or more designs into one blocked design for purposes of analysis.

Usage

```
djoin(design1, design2, ..., blkname = "Block", blocklev)
stdorder(design)
```

Arguments

`design1`	A `coded.data` object (must have been created by **rsm** 2.00 or higher).
`design2`	A `data.frame` (or `coded.data`) to be appended; or a call to a function that will create a design
`...`	Additional designs to be appended
`blkname`	Name to give to the blocking variable that distinguishes the designs that are joined
`blocklev`	Label to use in the blocking variable for the added design
`design`	A `coded.data` object to be displayed.

Details

`djoin` may be used to augment a design with all manner of other designs, including regular designs generated by `cube` and its relatives, `data.frames`, and other `coded.data` objects. The underlying paradigm is that each design joined is a separate block, and the order in which they are joined could matter.

It tries to do this in a smart way: The first design, `design1`, is required to be a `coded.data` object. If `design2` is a `data.frame`, and variables with the coded names are not present, it is automatically coded according to `design1`'s coding formulas. If `design2` is a `coded.data` object, and its coding formulas differ from those of `design1`, then `design1` is recoded with `design2`'s codings before the designs are joined. In both cases, any variables in `design2` not matched in `design1` are excluded, and any `design1` variables absent in `design2` are added with values of `NA`.

Value

`djoin` returns a `coded.data` object with the combined designs, and coding formulas from the last `coded.data` object added. The generated blocking variable will be a `factor`. The designs are sorted by blocks and `run.order` within blocks; and its `row.names` will be integers corresponding to this ordering.

The function `stdorder` sorts such data by block and `std.order` within block to display the designs in their pre-randomized order.

Author(s)

Russell V. Lenth

See Also

`cube`, `coded.data`, `bbd`

Examples

```
# Some existing data
CR1 <- coded.data(ChemReact1, x1 ~ (Time - 85)/5, x2 ~ (Temp - 175)/5)
# add the second part of the experiment; it gets coded automagically
djoin(CR1, ChemReact2)

# A new experiment in a different part of the design space
newdes <- cube(Yield ~ x1 + x2,  n0 = 3,
    coding = c(x1 ~ (Time - 70)/10, x2 ~ (Temp - 180)/5))
# Time passes ... we do the experiment and plug-in the observed Yield values
newdes$Yield <- rnorm(7, 75, 3) # these are our pretend results
combined <- djoin(CR1, newdes)
# Observe that the combined dataset is recoded to the new formulas
print(combined, decode = FALSE)

# List the new design in standard order
stdorder(newdes)
```

`FO` *Response-surface model components*

Description

Use of one of these functions in a model is how you specify the portion of the model that is to be regarded as a response-surface component.

Usage

```
FO (...)
TWI (..., formula)
PQ (...)
SO (...)
PE (...)
```

Arguments

`...`	The numerical predictors for the response surface, separated by commas.
`formula`	Alternative way to specify two-way interactions. Use `formula` or `...`, never both.

Details

Use `FO()` in the model formula in `rsm` to specify a first-order response surface (i.e., a linear function) in its arguments. Use `TWI()` to generate two-way interactions, and `PQ()` to generate pure quadratic terms (squares of the `FO()` terms). A call to `SO()` creates all terms in `FO()`, `TWI()`, and `PQ()` (in that order) for those variables. However, specifying `SO()` in a model formula in `rsm` will be replaced by the explicit sum of model terms, so that the `anova` table shows separate sums of squares. Other variables (such as blocks or factors) may be included in the model but should never be included in the arguments to `FO` or `SO`.

`PE` is used for fitting pure-error models. It should not be used in response-surface models. This function exists primarily for use by `loftest`, but could be useful in other linear-model contexts for fitting a model that interpolates the means at each distinct combination of argument values.

The `formula` argument in `TWI` can simplify specifying models where only certain interactions are included. For example, '`TWI(formula = ~x1:(x2+x3))`' is equivalent to '`TWI(x1,x2) + TWI(x1,x3)`'. The formula is expanded using `terms`, and then only the second-order terms are retained. If this results in only one term, an error condition is raised. This is necessary to prevent `rsm` from getting confused in identifying second-order terms.

Value

The functions `FO`, `TWI`, `PQ`, and `SO` return a matrix whose columns are the required predictors.

`PE` returns a `factor` whose levels are all the distinct combinations of arguments provided to the function.

Author(s)

Russell V. Lenth

References

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

See Also

`rsm`

Examples

```
### See 'rsm' help for examples of FO, TWI, etc

library(rsm)
### Test LOF for a regression model
ChemReact.lm <- lm(Yield ~ Time*Temp, data = ChemReact1)
PureError.lm <- update (ChemReact.lm, . ~ PE(Time,Temp))
anova (ChemReact.lm, PureError.lm)
```

`heli` *Paper Helicopter Data*

Description

A central composite design with 4 factors in 2 blocks. These data comprise a `coded.data` object.

Usage

```
heli
```

Format

A data frame with 30 observations on the following 7 variables. Each observation reflects the results of 10 replicated flights under the same experimental conditions.

`block` a factor with levels `1` `2`. Block 1 consists of 18 observations (a full factorial plus two center points). Block 2 consists of 12 observations – 8 axis points and 4 center points.

`x1` a numeric vector. Coded wing area, `x1 ~ (A - 12.4)/.6`

`x2` a numeric vector. Coded length ratio, `x2 ~ (R - 2.52)/.26`

`x3` a numeric vector. Coded body width, `x3 ~ (W - 1.25)/.25`

`x4` a numeric vector. Coded body length, `x4 ~ (L - 2)/.5`

`ave` a numeric vector. Average flight time, in csec.

`logSD` a numeric vector. 100*log(SD of times).

Source

Table 12.5 of Box, GEP, Hunter, JS, and Hunter, WG (2005) *Statistics for Experimenters* (2nd ed.), Wiley.

model.data — *Reconstruct data from a linear model*

Description

Create a data frame with just the variables in the formula in a `lm` object. This is comparable to `model.matrix` or `model.frame` except that factors, polynomials, transformations, etc. are not expanded.

Usage

```
model.data(lmobj, lhs = FALSE)
```

Arguments

`lmobj`	An object returned by `lm` or one of its relatives.
`lhs`	Boolean indicator of whether or not to include the variable(s) on the left-hand side of the model formula.

Details

This is an easy-to-use substitute for `get_all_vars`. The `formula`, `data`, and `subset` arguments, if present in `lmobj`'s call, affect the result appropriately.

Value

A data frame containing each of the variables referenced in the model formula.

Author(s)

Russell V. Lenth

References

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

See Also

`model.matrix`, `model.frame`

Examples

```
library(rsm)
trees.lm <- lm(log(Volume) ~ poly(log(Girth),3), data = trees, subset = 1:20)
model.frame(trees.lm)
model.data(trees.lm)
```

rsm *Response-surface regression*

Description

Fit a linear model with a response-surface component, and produce appropriate analyses and summaries.

Usage

```
rsm (formula, data, ...)

## S3 method for class 'rsm'
summary(object, adjust = rev(p.adjust.methods), ...)
## S3 method for class 'summary.rsm'
print(x, ...)

## S3 method for class 'rsm'
codings(object)

loftest (object)

canonical (object, threshold = 0.1*max.eigen)
xs (object, ...)
```

Arguments

`formula`	Formula to pass to `lm`. The model must include at least one `FO()`, `SO()`, `TWI()`, or `PQ()` term to define the response-surface portion of the model.
`data`	`data` argument to pass to `lm`.
`...`	In `rsm`, arguments that are passed to `lm`, `summary.lm`, or `canonical`, as appropriate. In `summary`, and `print`, additional arguments are passed to their generic methods.
`object`	An object of class `rsm`
`adjust`	Adjustment to apply to the P values in the coefficient matrix, chosen from among the available `p.adjust` methods in the **stats** package. The default is `"none"`.
`threshold`	Threshold for canonical analysis – see "Canonical analysis" below.
`x`	An object produced by `summary`

Details

In `rsm`, the model formula must contain at least an `FO` term; optionally, you can add one or more `TWI()` terms and/or a `PQ()` term. All variables that appear in `TWI` or `PQ` *must* be included in `FO`. For convenience, specifying `SO()` is the same as including `FO()`, `TWI()`, and `PQ()`, and is the safe, preferred way of specifying a full second-order model.

The variables in FO comprise the variables to consider in response-surface methods. They need not all appear in TWI and PQ terms; and more than one TWI term is allowed. For example, the following two model formulas are equivalent:

```
resp ~ Oper + FO(x1,x2,x3,x4) + TWI(x1,x2,x3) + TWI(x2,x3,x4) + PQ(x1,x3,x4)
resp ~ Oper + FO(x1,x2,x3,x4) + TWI(formula = ~x1*x2*x3 + x2*x3*x4) + PQ(x1,x3,x4)
```

The first version, however, creates duplicate `x2:x3` terms – which `rsm` can handle but there may be warning messages if it is subsequently used for predictions or plotted in `contour.lm`.

In `summary.rsm`, any `...` arguments are passed to `summary.lm`, except for `threshold`, which is passed to `canonical`.

Value

`rsm` returns an `rsm` object, which is a `lm` object with additional members as follows:

`order`	The order of the model: 1 for first-order, 1.5 for first-order plus interactions, or 2 for a model that contains square terms.
`b`	The first-order response-surface coefficients.
`B`	The matrix of second-order response-surface coefficients, if present.
`labels`	Labels for the response-surface terms. These make the summary much more readable.
`coding`	Coding formulas, if provided in the `codings` argument or if the `data` argument passed to `lm` is a `coded.data` object.

Summary and print methods

The `print` method for `rsm` objects just shows the call and the regression coefficints.

The `summary`method for `rsm` objects returns an object of class `summary.rsm`, which is an extension of the `summary.lm` class with these additional list elements:

sa Unit-length vector of the path of steepest ascent (first-order models only).

canonical Canonical analysis (second-order models only) from `canonical`

lof ANOVA table including lack-of-fit test.

coding Coding formulas in parent `rsm` object. Its `print` method shows the regression summary, followed by an ANOVA and lack-of-fit test. For first-order models, it shows the direction of steepest ascent (see `steepest`), and for second-order models, it shows the canonical analysis of the response surface.

Canonical analysis and stationary point

`canonical` returns a list with elements `xs`, the stationary point, and `eigen`, the eigenanalysis of the matrix **B** of second-order coefficients. Any eigenvalues less than `threshold` are taken to be zero, and a message is displayed. If this happens, the stationary point is determined using only the surviving eigenvectors, and stationary ridges or valleys are assumed to exist in their corresponding canonical directions. The default threshold is one tenth of the maximum eigenvalue, internally named `max.eigen`. Setting a small `threshold` may move the stationary point much farther from the origin.

When uncoded data are used, the canonical analysis and stationary point are not very meaningful and those results should probably be ignored. See '`vignette("rsm")`' for more details.

The function `xs` returns just the stationary point.

Other functions

`loftest` returns an `anova` object that tests the fitted model against a model that interpolates the means of the response-surface-variable combinations.

`codings` returns a `list` of coding formulas if the model was fitted to `coded.data`, or `NULL` otherwise.

emmeans support

Support is provided for the **emmeans** package: its `emmeans` and related functions work with special provisions for models fitted to coded data. The optional `mode` argument can have values of `"asis"` (the default), `"coded"`, or `"decoded"`. The first two are equivalent and simply return LS means based on the original model formula and the variables therein (raw or coded), without any conversion. When coded data were used and the user specifies `mode = "decoded"`, the user must specify results in terms of the decoded variables rather than the coded ones. See the illustration in the Examples section.

Author(s)

Russell V. Lenth

References

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

See Also

`FO`, `SO`, `lm`, `summary`, `coded.data`

Examples

```
library(rsm)
CR <- coded.data (ChemReact, x1~(Time-85)/5, x2~(Temp-175)/5)

### 1st-order model, using only the first block
CR.rs1 <- rsm (Yield ~ FO(x1,x2), data=CR, subset=1:7)
summary(CR.rs1)

### 2nd-order model, using both blocks
CR.rs2 <- rsm (Yield ~ Block + SO(x1,x2), data=CR)
summary(CR.rs2)

### Example of a rising-ridge situation from Montgomery et al, Table 6.2
```

```
RRex <- ccd(Response ~ A + B, n0 = c(0, 3), alpha = "face",
            randomize = FALSE, oneblock = TRUE)
RRex$Response <- c(52.3, 5.3, 46.7, 44.2, 58.5, 33.5, 32.8, 49.2, 49.3, 50.2, 51.6)
RRex.rsm <- rsm(Response ~ SO(A,B), data = RRex)
canonical(RRex.rsm)  # rising ridge is detected
canonical(RRex.rsm, threshold = 0)  # xs is far outside of the experimental region

## Not run:
# Illustration of emmeans support
emmeans::emmeans(CR.rs2, ~ x1 * x2, mode = "coded",
        at = list(x1 = c(-1, 0, 1), x2 = c(-2, 2)))

# The following will yield the same results, but based on the decoded data
emmeans::emmeans(CR.rs2, ~ Time * Temp, mode = "decoded",
        at = list(Time = c(80, 85, 90), Temp = c(165, 185)))

## End(Not run)
```

steepest — *Steepest-ascent methods for response surfaces*

Description

These functions provide the path of steepest ascent (or descent) for a fitted response surface produced by `rsm`.

Usage

```
steepest (object, dist = seq(0, 5, by = .5), descent = FALSE)
canonical.path(object, which = ifelse(descent, length(object$b), 1),
                 dist = seq(-5, 5, by = 0.5), descent = FALSE, ...)
```

Arguments

`object`	`rsm` object to be analyzed.
`dist`	Vector of desired distances along the path of steepest ascent or descent. In `steepest`, these must all be non-negative; in `canonical.path`, you may want both positive and negative values, which specify opposite directions from the stationary point.
`descent`	Set this to `TRUE` to obtain the path of steepest descent, or `FALSE` to obtain the path of steepest ascent. This value is ignored in `canonical.path` if `which` is specified.
`which`	Which canonical direction (eigenvector) to use.
`...`	Optional arguments passed to `canonical`. Currently this includes only `threshold`.

Details

`steepest` returns the linear path of steepest ascent for first-order models, or a path obtained by ridge analysis (see Draper 1963) for second-order models. In either case, the path begins at the origin.

`canonical.path` applies only to second-order models (at least a `TWI` term present). It determines a linear path along one of the canonical variables, originating at the stationary point (not the origin). We need to specify which canonical variable to use. The eigenvalues obtained in the canaonical analysis are always in decreasing order, so the first canonical direction will be the path of steepest ascent (or slowest descent, if all eigenvalues are negative) from the stationary point, and the last one will be the path of steepest descent (or slowest ascent, if all eigenvalues are positive). These are the defaults for `which` when `descent=FALSE` and `descent=TRUE` respectively.

All eigenvalues less (in absolute value than) `threshold` are taken to be zero. Increasing this threshold may bring the stationary point, and hence the canonical path, much closer to the design center, and thus less extrapolation.

With either function, the path in uncoded units depends on how the data are coded. Accordingly, it is important to code the predictor variables appropriately before fitting the response-surface model. See `coded.data` and its relatives for more information.

Value

A `data.frame` of points along the path of steepest ascent (or descent). For `steepest`, this path originates from the center of the experiment; for `canonical.path`, it starts at the stationary point. If coding information is available, the data frame also includes the uncoded values of the variables.

For first-order response surfaces, only `steepest` may be used; the path is linear in that case. For second-order surfaces, `steepest` uses ridge analysis, and the path may be curved.

Note

Take careful note of the fitted values along the outputted path (labeled `yhat`). For example, if the stationary point is a maximum (all eigenvalues negative), the fitted values from `steepest` will increase as far as the stationary point, then they will decrease as we proceed along what is now the path of slowest descent.

Author(s)

Russell V. Lenth

References

Draper, NR (1963), “Ridge analysis of response surfaces”, *Technometrics*, 5, 469–479.

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

See Also

`rsm`, `coded.data`

Examples

```
library(rsm)
heli.rsm = rsm (ave ~ block + SO(x1, x2, x3, x4), data = heli)

steepest(heli.rsm)

canonical.path(heli.rsm)
```

---

varfcn *Display the scaled variance function for a design*

Description

This function computes the scaled variance function for a design, based on a specified model. Options include plotting separate curves for each of several directions from the center, or a contour plot for two of the design factors.

Usage

```
varfcn(design, formula, dist = seq(0, 2, by = 0.1), vectors, contour = FALSE,
       plot = TRUE, main, ...)
```

Arguments

`design`	A `data.frame` or `coded.data` object
`formula`	The model formula for which to compute the variance function
`dist`	Vector of distances from the origin at which to compute the scaled variance
`vectors`	A `data.frame` of design variables. Each nonzero row specifies a direction in which to calculate the scaled variance.
`contour`	A logical variable. If `TRUE`, a contour plot is produced; if `FALSE`, curves are plotted for each direction in `vectors`.
`plot`	A logical variable. If `TRUE`, a plot is produced.
`main`	Title for the plot. The default is constructed based on the name of `design` and `formula`.
`...`	Other arguments passed to the `plot` or `contour` functions.

Details

The scaled prediction variance at a particular design point is the variance of the predicted value, multiplied by the sample size *N*, and divided by the error variance. (See, for example, Montgomery *et al.*, Section 8.2.1). It depends on the design point, but for a symmetric design, it depends only on the distance from the origin and the direction. This function provides a simple way to examine the variance function directly. (There are other more sophisticated methods available that integrate-out the direction, for example `Vdgraph` in the `Vdgraph` package.)

If vectors is not specified and contour==FALSE, the function generates default directions along one axis, and on a diagonal through a corner in each dimension. For example, with four design variables, the default directions are (1,0,0,0), (1,1,0,0), (1,1,1,0), and (1,1,1,1). The graph produced shows how the scaled variance changes along each of these vectors, for the distances provided. In a rotatable design, these curves will all be the same.

When countour==TRUE, only the ordering of columns in vectors matters. A grid is constructed over the distance range for the first two variables in vectors. The design points are also plotted for reference, with different symbol sizes depending on replications. When there are more than two response-surface predictors, the contour plot may be misleading, as it does not display what happens as one simultaneously varies three or more variables.

Value

The function invisibly returns a data.frame containing the data that was (or would have been) plotted.

Author(s)

Russell V. Lenth

References

Myers, RH Montgomery DC, and Anderson-Cook CM (2009) *Response Surface Methodology* (3rd ed.), Wiley.

See Also

Vdgraph

Examples

```
des = ccd(~ x1 + x2 + x3, alpha = 1.5, block = Phase ~ x1*x2*x3, randomize=FALSE)
varfcn(des, ~ Phase + SO(x1, x2, x3))
varfcn(des, ~ Phase + SO(x1, x2, x3), contour=TRUE)

# 10 random directions
dirs = data.frame(x3=rnorm(10), x2=rnorm(10), x1=rnorm(10))
varfcn(des, ~ Phase + SO(x1, x2, x3), vectors = dirs)

# exclude some points to make it more interesting
lost = c(1,2,3,5,8,13,21)
varfcn(des[-lost, ], ~ Phase + SO(x1, x2, x3), contour=TRUE)

# different plot due to order of columns
varfcn(des[-lost, ], ~ Phase + SO(x1, x2, x3), vectors = dirs, contour=TRUE)
```

Index