Package ‘docopulae’

October 26, 2018

Title Optimal Designs for Copula Models
Version 0.4.0
Date 2018-10-26
Author Andreas Rappold [aut, cre]
Maintainer Andreas Rappold <arappold@gmx.at>

Description A direct approach to optimal designs for copula models based on
the Fisher information. Provides flexible functions for building joint PDFs,
evaluating the Fisher information and finding optimal designs. It includes an
extensible solution to summation and integration called 'nint', functions for
transforming, plotting and comparing designs, as well as a set of tools for
common low-level tasks.

Depends R (>= 3.1.2)
Imports graphics, grDevices, methods, stats, utils
Suggests copula, numDeriv, Deriv (>= 3.8.5), cubature, SparseGrid,
mvtnorm, testthat

URL http://www.tandfonline.com/doi/full/10.1080/02331888.2015.1111892
      https://github.com/arappold/docopulae

BugReports https://github.com/arappold/docopulae/issues
License MIT + file LICENSE
LazyData true
NeedsCompilation yes
Encoding UTF-8
RoxygenNote 6.1.0
Repository CRAN
Date/Publication 2018-10-26 10:20:04 UTC
R topics documented:

buildf .................................................. 3
Defficiency ............................................. 6
DerivLogf ............................................... 7
design .................................................... 8
docopulae ............................................... 9
Dsensitivity ........................................... 9
fisherI ................................................... 11
getM ...................................................... 12
grow.grid ................................................ 12
integrateA ............................................... 13
nint_ERROR .......................................... 14
nint_expandSpace ..................................... 14
nint_funcDim ........................................ 15
nint_gridDim .......................................... 16
nint_integrate ......................................... 16
nint_integrateNCube .................................. 18
nint_integrateNFunc .................................. 20
nint_intvDim .......................................... 22
nint_scatDim ......................................... 23
nint_space ........................................... 24
nint_tanTransform .................................. 25
nint_transform ....................................... 26
nint>Type ............................................. 29
nint_validateSpace ................................. 30
numDerivLogf ....................................... 31
param ..................................................... 32
plot.desigh .......................................... 35
print.nint_space .................................... 36
reduce .................................................... 37
rowmatch .............................................. 38
roworder ............................................... 39
rowsduplicated ....................................... 39
seq1 ...................................................... 40
update.param ........................................ 41
wDefficiency ......................................... 42
wDsensitivity ....................................... 43
Wynn ...................................................... 44

Index 45
buildf

Build probability density or mass Function

Description

buildf builds a joint probability density or mass function from marginal distributions and a copula.

Usage

buildf(margins, continuous, copula, parNames = NULL, simplifyAndCache = T)

Arguments

margins either

- function(y, theta, ...), where theta is a list of parameters. It shall return a column matrix of two, the probability densities and cumulative distributions.
- a list of pairs of expressions, each named "pdf" and "cdf", the probability density and cumulative distribution.

continuous TRUE if margins are continuous. See details.

copula if margins is

- a function then either a copula object from package copula or function(u, theta, ...), a probability density function if continuous else a cumulative distribution function.
- a list then either a copula object from package copula which contains distribution expressions or an expression for the probability density if continuous else the cumulative distribution which uses u1,u2,...

parNames if

- (optional) margins is a function and copula is a copula object then a vector of names or indices, the sequence of copula parameters in theta. 0 or "" identifies copula parameters to skip.
- margins is a list and copula is a copula object then a named list of names or indices, mapping parameters in theta to copula parameter variables. See copula@exprdist.

simplifyAndCache (if margins is a list) simplify and cache the result using Simplify and Cache from package Deriv if available.

Details

Please note that expressions are not validated.

If continuous is FALSE, dimensionality shall be 2 and both dimensions shall be discrete. The joint probability mass is defined by

\[ C(F_1(y_1), F_2(y_2)) - C(F_1(y_1 - 1), F_2(y_2)) - C(F_1(y_1), F_2(y_2 - 1)) + C(F_1(y_1 - 1), F_2(y_2 - 1)) \]
where $C$, $F_1$, and $F_2$ depend on $\theta$ and $y_i \geq 0$.

**Value**

`buildf` returns `function(y, theta, ...)`, the joint probability density or mass function.

**See Also**

`copula`, `Simplify.Cache, numDerivLogf, DerivLogf, fisherI`

**Examples**

```r
## for an actual use case see examples for `param`
library(copula)
library(mvtnorm)

## build bivariate normal
margins = function(y, theta) {
  mu = c(theta$mu1, theta$mu2)
  cbind(dnorm(y, mean=mu, sd=1), pnorm(y, mean=mu, sd=1))
}
copula = normalCopula()

# args: function, copula object, parNames
f1 = buildf(margins, TRUE, copula, parNames='alpha')
f1 # uses theta[['alpha']] as copula parameter

## evaluate and plot
theta = list(mu1=2, mu2=-3, alpha=0.4)
y1 = seq(0, 4, length.out=51)
y2 = seq(-5, -1, length.out=51)
v1 = outer(y1, y2, function(z1, z2) apply(cbind(z1, z2), 1, f1, theta))
str(v1)
contour(y1, y2, v1, main='f1', xlab='y1', ylab='y2')

## compare with bivariate normal from `mvtnorm`
copula(parameters = theta$alpha)
v = outer(y1, y2, function(yy1, yy2)
  dmvnorm(cbind(yy1, yy2),
          mean=c(theta$mu1, theta$mu2),
          sigma=getSigma(copula)))
all.equal(v1, v)

## build bivariate pdf with normal margins and Clayton copula
margins = list(list(pdf=quote(dnorm(y[1], theta$mu1, 1)),
                  cdf=quote(pnorm(y[1], theta$mu1, 1))),
               list(pdf=quote(dnorm(y[2], theta$mu2, 1)),
                    cdf=quote(pnorm(y[2], theta$mu2, 1))))
copula = claytonCopula()
```
```r
# args: list, copula object, parNames
f2 = buildf(margins, TRUE, copula, list(alpha='alpha1'))
f2

## evaluate and plot
theta = list(mu1=2, mu2=-3, alpha1=2)
y1 = seq(0, 4, length.out=51)
y2 = seq(-5, -1, length.out=51)
v2 = outer(y1, y2, function(z1, z2) apply(cbind(z1, z2), 1, f2, theta))
str(v2)
contour(y1, y2, v2, main='f2', xlab='y1', ylab='y2')

## build alternatives
cexpr = substituteDirect(copula@exprdist$pdf,
  list(alpha=quote(theta$alpha1)))
# args: list, expression
f3 = buildf(margins, TRUE, cexpr) # equivalent to f2
f3

margins = function(y, theta) {
  mu = c(theta$mu1, theta$mu2)
  cbind(dnorm(y, mean=mu, sd=1), pnorm(y, mean=mu, sd=1))
}
# args: function, copula object, parNames
f4 = buildf(margins, TRUE, copula, 'alpha1')
f4

cpdf = function(u, theta) {
  copula@parameters = theta$alpha1
  dCopula(u, copula)
}
# args: function, function
f5 = buildf(margins, TRUE, cpdf) # equivalent to f4
f5

# args: function, copula object
copula@parameters = 2
f6 = buildf(margins, TRUE, copula)
f6 # uses copula@parameters
cpdf = function(u, theta) dCopula(u, copula)
# args: function, function
f7 = buildf(margins, TRUE, cpdf) # equivalent to f6
f7

## compare all
vv = lapply(list(f3, f4, f5, f6, f7), function(f)
  outer(y1, y2, function(z1, z2) apply(cbind(z1, z2), 1, f, theta)))
sapply(vv, all.equal, v2)
```
Defficiency computes the D-, D_s or D_A-efficiency measure for a design with respect to a reference design.

Usage

Defficiency(des, ref, mod, A = NULL, parNames = NULL)

Arguments

des a design.
ref a design, the reference.
mod a model.
A for
• D-efficiency: NULL
• D_s-efficiency: a vector of names or indices, the subset of parameters of interest.
• D_A-efficiency: either
  – directly: a matrix without row names.
  – indirectly: a matrix with row names corresponding to the parameters.
parNames a vector of names or indices, the subset of parameters to use. Defaults to the parameters for which the Fisher information is available.

Details

Indices supplied to argument A correspond to the subset of parameters defined by argument parNames. D efficiency is defined as
\[
\frac{1}{n} \left( \frac{|M(\xi, \bar{\theta})|}{|M(\xi^*, \bar{\theta})|} \right)^{1/n}
\]
and D_A efficiency as
\[
\left( \frac{|A^T M(\xi, \bar{\theta})^{-1} A|^{-1}}{|A^T M(\xi^*, \bar{\theta})^{-1} A|^{-1}} \right)^{1/s}
\]

Value

Defficiency returns a single numeric.

See Also
design, param, wDefficiency
Examples

### see examples for param

---

**DerivLogf** *Build Derivative Function for Log f*

**Description**

DerivLogf/Deriv2Logf builds a function that evaluates to the first/second derivative of \( \log(f(y, \theta, \ldots)) \) with respect to \( \theta[i]/\theta[i] \) and \( \theta[j] \).

**Usage**

DerivLogf(f, parNames, preSimplify = T, ...)
Deriv2Logf(f, parNames, preSimplify = T, ...)

**Arguments**

- \( f \) function\((y, \theta, \ldots)\), where \( \theta \) is a list of parameters.
- \( \text{parNames} \) a vector of names or indices, the subset of parameters to use.
- \( \text{preSimplify} \) simplify the body of \( f \) using functions from package Deriv.
- ... other arguments passed to Deriv from package Deriv.

**Details**

While numDerivLogf relies on the package numDeriv and therefore uses finite differences to evaluate the derivatives, DerivLogf utilizes the package Deriv to build sub functions for each parameter in parNames. The same is true for Deriv2Logf.

**Value**

DerivLogf returns function\((y, \theta, i, \ldots)\) which evaluates to the first derivative of \( \log(f(y, \theta, \ldots)) \) with respect to \( \theta[i] \). The attribute "d" contains the list of sub functions.

Deriv2Logf returns function\((y, \theta, i, j, \ldots)\) which evaluates to the second derivative of \( \log(f(y, \theta, \ldots)) \) with respect to \( \theta[i] \) and \( \theta[j] \). The attribute "d2" contains the list of sub functions.

**See Also**

Deriv, Deriv in package Deriv, buildf, numDerivLogf, fisher1
Examples

## see examples for param
## mind the gain regarding runtime compared to numDeriv

---

<table>
<thead>
<tr>
<th><code>design</code></th>
<th><code>Design</code></th>
</tr>
</thead>
</table>

## Description

`design` creates a custom design object.

## Usage

```r
design(x, w, tag = list())
```

## Arguments

- `x`: a row matrix of points.
- `w`: a vector of weights. Length shall be equal to the number of rows in `x` and sum shall be equal to `Q`.
- `tag`: a list containing additional information about the design.

## Value

`design` returns an object of class "design". An object of class "design" is a list containing at least this function’s arguments.

## See Also

`Wynn`, `reduce`, `getM`, `plot.desigh`, `Deficiency`, `update.param`

## Examples

## see examples for param
Description

A direct approach to optimal designs for copula models based on the Fisher information. Provides flexible functions for building joint PDFs, evaluating the Fisher information and finding optimal designs. It includes an extensible solution to summation and integration called ‘nint’, functions for transforming, plotting and comparing designs, as well as a set of tools for common low-level tasks.

Details

This package builds upon the theoretical result on optimal designs for copula models developed by Elisa Perrone and Werner G. Müller. In their paper named ‘Optimal designs for copula models’ they introduce an equivalence theorem of Kiefer-Wolfowitz type for D-optimality along with examples and the proof. The proof for D_A-optimality is analogous and is mentioned in an upcoming paper currently under double blind review.

References


See Also

D sensitivity

D sensitivity

D Sensitivity

Description

D sensitivity builds a sensitivity function for the D-, D_s or D_A-optimality criterion which relies on defaults to speed up evaluation. Wynn for instance requires this behaviour/protocol.

Usage

D sensitivity(A = NULL, parNames = NULL, defaults = list(x = NULL, desw = NULL, desx = NULL, mod = NULL))
Dsensitivity

Arguments

A  for
• D-optimality: NULL
• D_s-optimality: a vector of names or indices, the subset of parameters of interest.
• D_A-optimality: either
  – directly: a matrix without row names.
  – indirectly: a matrix with row names corresponding to the parameters.
parNames  a vector of names or indices, the subset of parameters to use. Defaults to the parameters for which the Fisher information is available.
defaults  a named list of default values. The value NULL is equivalent to absence.

Details

Indices and rows of an unnamed matrix supplied to argument A correspond to the subset of parameters defined by argument parNames.

For efficiency reasons the returned function won’t complain about missing arguments immediately, leading to strange errors. Please ensure that all arguments are specified at all times. This behaviour might change in future releases.

Value

Dsensitivity returns function(x=NULL, desw=NULL, desx=NULL, mod=NULL), the sensitivity function. It’s attributes contain this function’s arguments.

References


See Also
docopulae, param, wDsensitivity, Wynn, plot.desigh

Examples

## see examples for param
Description

fisherI utilizes nint_integrate to evaluate the Fisher information.

Usage

fisherI(ff, theta, parNames, yspace, ...)

Arguments

- `ff`: either
  - function(y, theta, i, j, ...) which evaluates to the inner part of the expectation integral/sum.
  - list(f=function(y, theta, ...), d2logf=function(y, theta, i, j, ...)) (recommended)
  - list(f=function(y, theta, ...), dlogf=function(y, theta, i, ...))

where f is the joint probability density function and dlogf/d2logf is the first/second derivative of log(f) with respect to theta[i]/theta[i] and theta[j].

- `theta`: the list of parameters.
- `parNames`: a vector of names or indices, the subset of parameters to use.
- `yspace`: a space, the support of y.
- `...`: other arguments passed to ff.

Details

If ff is a list, it shall contain dlogf xor d2logf.

Value

fisherI returns a named matrix, the Fisher information.

See Also

buildf, numDerivLogf, DerivLogf, nint_space, nint_transform, nint_integrate, param

Examples

```R
## see examples for param
```
getM  Get Fisher Information

Description

getM returns the Fisher information corresponding to a model and a design.

Usage

getM(mod, des)

Arguments

mod  a model.
des  a design.

Value

getM returns a named matrix, the Fisher information.

See Also

param, design

Examples

## see examples for param

grow.grid  Grow Grid

Description

grow.grid creates a data frame like expand.grid. The order of rows is adjusted to represent a growing grid with respect to resolution.

Usage

grow.grid(x, random = T)

Arguments

x  a list of vectors.
random  TRUE if order of rows within each level of resolution should be random.
integrateA

Value

grow.grid returns a data frame like expand.grid.

See Also

update.param

integrateA  Integrate Alternative

Description

integrateA is a tolerance wrapper for integrate. It allows integrate to reach the maximum number of subdivisions.

Usage

integrateA(f, lower, upper, ..., subdivisions = 100L,
    rel.tol = .Machine$double.eps^0.25, abs.tol = rel.tol,
    stop.on.error = TRUE, keep.xy = FALSE, aux = NULL)

Arguments

f, lower, upper, ..., subdivisions, rel.tol, abs.tol, stop.on.error, keep.xy, aux

see integrate.

Details

See integrate.

See Also

integrate

Examples

f = function(x) ifelse(x < 0, cos(x), sin(x))
#curve(f(x), -1, 1)
try(integrate(f, -1, 1, subdivisions=1)$value)
integrateA(f, -1, 1, subdivisions=1)$value
integrateA(f, -1, 1, subdivisions=2)$value
integrateA(f, -1, 1, subdivisions=3)$value
**nint_expandSpace**

**Space Validation Errors**

**Description**

Error codes for space validation.

**Usage**

```
nint_ERROR_DIM_TYPE # = -1001
nint_ERROR_SCATTER_LENGTH # = -1002
nint_ERROR_SPACE_TYPE # = -1003
nint_ERROR_SPACE_DIM # = -1004
```

**Format**

integer

**Details**

- **nint_ERROR_DIM_TYPE**: dimension type attribute does not exist or is invalid.
- **nint_ERROR_SCATTER_LENGTH**: scatter dimensions have different lengths.
- **nint_ERROR_SPACE_TYPE**: object not of type "nint_space".
- **nint_ERROR_SPACE_DIM**: subspaces have different number of dimensions.

**See Also**

`nint_validateSpace`, `nint_space`

---

**Expand Space**

**Description**

`nint_expandSpace` expands a space or list structure of spaces to a list of true subspaces.

**Usage**

```
nint_expandSpace(x)
```
**nint_funcDim**

**Arguments**

x  
a space or list structure of spaces.

**Value**

nint_expandSpace returns a list of spaces. Each space is a true subspace.

**See Also**

nint_space

**Examples**

```
s = nint_space(list(nint_intvDim(1, 2),
                  nint_intvDim(3, 4)),
               list(nint_intvDim(-Inf, 0),
                    nint_gridDim(c(0)),
                    nint_intvDim(0, Inf))
)
s
nint_expandSpace(s)
```

---

**nint_funcDim**  
*Function Dimension*

**Description**

nint_funcDim defines a functionally dependent dimension. It shall depend solely on the previous dimensions.

**Usage**

nint_funcDim(x)

**Arguments**

x  
function(x), where x is the partially realized point in the space. It shall return an object of type nint_intvDim or a vector.

**Details**

Obviously if x returns an object of type nint_intvDim the dimension is continuous, and discrete otherwise.

As the argument to x is only partially defined the user has to ensure that the function solely depends on values up to the current dimension.

**Value**

nint_scatDim returns its argument with the dimension type attribute set to nint_TYPE_FUNC_DIM.
**nint_integrate**

**Description**

*nint_integrate* performs summation and integration of a scalar-valued function over a space or list structure of spaces.

**Usage**

```R
nint_integrate(f, space, ...)
```

**See Also**

*nint_TYPE, nint_space*

---

**nint_gridDim**

**Grid Dimension**

**Description**

*nint_gridDim* is defined by a sequence of values. Together with other grid dimensions it defines a dense grid.

**Usage**

```R
nint_gridDim(x)
```

**Arguments**

- **x**: a vector of any type.

**Details**

Imagine using `expand.grid` to create a row matrix of points.

**Value**

*nint_scatDim* returns its argument with the dimension type attribute set to `nint_TYPE_GRID_DIM`.

**See Also**

*nint_TYPE, nint_space*
**Arguments**

- **f**: the scalar-valued function (integrand) to be integrated.
- **space**: a space or list structure of spaces.
- **...**: other arguments passed to `f`.

**Details**

`nint_integrate` uses `nint_integrateNCube` and `nint_integrateNFunc` to handle interval and function dimensions. See their help pages on how to deploy different solutions.

The order of dimensions is optimized for efficiency. Therefore interchangeability (except for function dimensions) is assumed.

**Value**

`nint_integrate` returns a single numeric.

**See Also**

`nint_space`, `nint_transform`, `nint_integrateNCube`, `nint_integrateNFunc`, `fisherI`

**Examples**

```r
## discrete
## a) scatter
s = nint_space(nint_scatDim(1:3),
    nint_scatDim(c(0, 2, 5)))
s
## (1, 0), (2, 2), (3, 5)
nint_integrate(function(x) abs(x[1] - x[2]), s) # 1 + 0 + 2 == 3

## b) grid
s = nint_space(nint_gridDim(1:3),
    nint_gridDim(c(0, 2, 5)))
s
## (1, 0), (1, 2), (1, 5), (2, 0), ..., (3, 2), (3, 5)
nint_integrate(function(x) ifelse(sum(x) < 5, 1, 0), s) # 5

## continuous
## c)
s = nint_space(nint_intvDim(1, 3),
    nint_intvDim(1, Inf))
s
nint_integrate(function(x) 1/x[2]**2, s) # 2

## d) infinite, no transform
s = nint_space(nint_intvDim(-Inf, Inf))
nint_integrate(sin, s) # 0

## e) infinite, transform
```
Description

Interface to the integration over interval dimensions.

Usage

nint_integrateNCube(f, lowerLimit, upperLimit, ...)
nint_integrateNCube_integrate(integrate)
nint_integrateNCube_cubature(adaptIntegrate)
nint_integrateNCube

nint_integrateNCube_SparseGrid(createIntegrationGrid)

Arguments

- integrate: function(f, lowerLimit, upperLimit, ...) which calls integrate.
- adaptIntegrate: function(f, lowerLimit, upperLimit, ...) which calls cubature::adaptIntegrate.
- createIntegrationGrid: function(dimension) which calls SparseGrid::createIntegrationGrid.
- f: the scalar-valued wrapper function to be integrated.
- lowerLimit: the lower limits of integration.
- upperLimit: the upper limits of integration.
- ...: other arguments passed to f.

Details

nint_integrate uses nint_integrateNCube to handle interval dimensions. See examples below on how to deploy different solutions.

The function built by nint_integrateNCube_integrate calls integrate (argument) recursively. The number of function evaluations therefore increases exponentially with the number of dimensions \((\text{subdivisions} \times 21) ^ D\) if integrate, the default, is used. At the moment it is the default method because no additional package is required. However, you most likely want to consider different solutions.

The function built by nint_integrateNCube_cubature is a trivial wrapper for cubature::adaptIntegrate.

The function built by nint_integrateNCube_SparseGrid is an almost trivial wrapper for SparseGrid::createIntegrationGrid. It scales the grid to the integration region.

Value

nint_integrateNCube returns a single numeric.

nint_integrateNCube_integrate returns a recursive implementation for nint_integrateNCube based on one dimensional integration.

nint_integrateNCube_cubature returns a trivial implementation for nint_integrateNCube indirectly based on cubature::adaptIntegrate.

nint_integrateNCube_SparseGrid returns an implementation for nint_integrateNCube indirectly based on SparseGrid::createIntegrationGrid.

See Also

- nint_integrate
- integrateA, integrate
- adaptIntegrate in package cubature
- createIntegrationGrid in package SparseGrid
Examples

```r
## integrate with defaults (stats::integrate)
nint_integrate(sin, nint_space(nint_intvDim(pi/4, 3*pi/4)))

dfltnCube = nint_integrateNCube

## prepare for integrateA
ncube = function(f, lowerLimit, upperLimit, ...) {
  cat('using integrateA\n')
  integrateA(f, lowerLimit, upperLimit, ..., subdivisions=2)
}
ncube = nint_integrateNCube_integrate(ncube)
unlockBinding('nint_integrateNCube', environment(nint_integrate))
assign('nint_integrateNCube', ncube, envir=environment(nint_integrate))

## integrate with integrateA
nint_integrate(sin, nint_space(nint_intvDim(pi/4, 3*pi/4)))

## prepare for cubature
ncube = function(f, lowerLimit, upperLimit, ...) {
  cat('using cubature\n')
  r = cubature::adaptIntegrate(f, lowerLimit, upperLimit, ..., maxEval=1e3)
  return(r$integral)
}
unlockBinding('nint_integrateNCube', environment(nint_integrate))
assign('nint_integrateNCube', ncube, envir=environment(nint_integrate))

## integrate with cubature
nint_integrate(sin, nint_space(nint_intvDim(pi/4, 3*pi/4)))

## prepare for SparseGrid
ccube = function(dimension) {
  cat('using SparseGrid\n')
  SparseGrid::createIntegrationGrid('GQU', dimension, 7)
}
ccube = nint_integrateNCube_SparseGrid(ncube)
unlockBinding('nint_integrateNCube', environment(nint_integrate))
assign('nint_integrateNCube', ncube, envir=environment(nint_integrate))

## integrate with SparseGrid
nint_integrate(sin, nint_space(nint_intvDim(pi/4, 3*pi/4)))

assign('nint_integrateNCube', dfltnCube, envir=environment(nint_integrate))
```
**Description**

Inferface to the integration over function dimensions.

**Usage**

```r
nint_integrateNFunc(f, funcs, x0, i0, ...)
nint_integrateNFunc_recursive(integrate1)
```

**Arguments**

- `integrate1` function(`f`, `lowerLimit`, `upperLimit`, ...) which performs one dimensional integration.
- `f` the scalar-valued wrapper function to be integrated.
- `funcs` the list of function dimensions.
- `x0` the partially realized point in the space.
- `i0` the vector of indices of function dimensions in the space.
- `...` other arguments passed to `f`.

**Details**

`nint_integrate` uses `nint_integrateNFunc` to handle function dimensions. See examples below on how to deploy different solutions.

The function built by `nint_integrateNFunc_recursive` directly sums over discrete dimensions and uses `integrate1` otherwise. In conjunction with `integrateA` this is the default.

**Value**

- `nint_integrateNFunc` returns a single numeric.
- `nint_integrateNFunc_recursive` returns a recursive implementation for `nint_integrateNFunc`.

**See Also**

- `nint_integrate`
- `integrateA`

**Examples**

```r
dfltNFunc = nint_integrateNFunc

## area of circle
s = nint_space(
    nint_intvDim(-1, 1),
    nint_funcDim(function(x) nint_intvDim(c(-1, 1) * sin(acos(x[1]))) )
)
nint_integrate(function(x) 1, s) # pi

## see nint_integrate's examples for more sophisticated integrals
```
```r
defines a fixed interval. The bounds may be (negative) Inf.
Usage
nint_intvDim(x, b = NULL)
Arguments
x either a single numeric, the lower bound, or a vector of length 2, the lower and upper bound.
b the upper bound if x is the lower bound.
Value

\texttt{nint\_intvDim} returns a vector of length 2 with the dimension type attribute set to \texttt{nint\_TYPE\_INTV\_DIM}.

See Also

\texttt{nint\_TYPE}, \texttt{nint\_space}

\begin{tabular}{ll}
\texttt{nint\_scatDim} & \textit{Scatter Dimension} \\
\end{tabular}

Description

\texttt{nint\_scatDim} is defined by a sequence of values. Together with other scatter dimensions it defines a sparse grid.

Usage

\texttt{nint\_scatDim(x)}

Arguments

\begin{itemize}
\item \texttt{x} \hspace{1cm} a vector of any type.
\end{itemize}

Details

Imagine using \texttt{cbind} to create a row matrix of points.

Value

\texttt{nint\_scatDim} returns its argument with the dimension type attribute set to \texttt{nint\_TYPE\_SCAT\_DIM}.

See Also

\texttt{nint\_TYPE}, \texttt{nint\_space}
nint_space

Description

nint_space defines an n-dimensional space as a list of dimensions. A space may consist of subspaces. A space without subspaces is called true subspace.

Usage

nint_space(...)

Arguments

... dimensions each of which may be an actual dimension object or a list structure of dimension objects.

Details

If a space contains at least one list structure of dimension objects it consists of subspaces. Each subspace is then defined by a combination of dimension objects along the dimensions. See nint_expandspace on how to expand a space to true subspaces.

Value

nint_space returns an object of class "nint_space". An object of class "nint_space" is an ordered list of dimension objects.

See Also

nint_scatDim, nint_gridDim, nint_intvDim, nint_funcDim, nint_integrate, nint_validateSpace, nint_expandspace, fisherI

Examples

```r
s = nint_space(nint_gridDim(seq(1, 3, 0.9)),
                nint_scatDim(seq(2, 5, 0.8)),
                nint_intvDim(-Inf, Inf),
                nint_funcDim(function(x) nint_intvDim(0, x[1])),
                list(nint_gridDim(c(0, 10)),
                     list(nint_intvDim(1, 7)))
            )

s
```
**Description**

`nint_tanTransform` creates the transformation \( g(x) = \text{atan}((x - \text{center})/\text{scale}) \) to be used in `nint_transform`.

**Usage**

```r
nint_tanTransform(center, scale, dIdcs = NULL)
```

**Arguments**

- `center`, `scale` see \( g(x) \).
- `dIdcs` an integer vector of indices, the dimensions to transform.

**Value**

`nint_tanTransform` returns a named list of two functions "g" and "giDgi" as required by `nint_transform`.

**See Also**

`nint_transform`

**Examples**

```r
mu = 1e0
sigma = mu/3
f = function(x) dnorm(x, mean=mu, sd=sigma)
space = nint_space(nint_intvDim(-Inf, Inf))

tt = nint_transform(f, space, list(nint_tanTransform(0, 1, dIdcs=1)))
tt$space
ff = Vectorize(tt$f); curve(ff(x), tt$space[[1]][1], tt$space[[2]][2])

nint_integrate(tt$f, tt$space) # should return 1

# same with larger mu
mu = 1e4
sigma = mu/3
f = function(x) dnorm(x, mean=mu, sd=sigma)

tt = nint_transform(f, space, list(nint_tanTransform(0, 1, dIdcs=1)))
ff = Vectorize(tt$f); curve(ff(x), tt$space[[1]][1], tt$space[[1]][2])

try(nint_integrate(tt$f, tt$space)) # integral is probably divergent

# same with different transformation
```
\texttt{tt = nint_transform(f, space, list(nint_transform(mu, sigma, dIdcs=1)))}
\texttt{ff = Vectorize(tt$f); curve(ff(x), tt$space[[1]][1], tt$space[[1]][2])}

\texttt{nint_integrate(tt$f, tt$space) \# should return 1}

\section*{nint_transform \quad Transform Integral}

\subsection*{Description}

\texttt{nint_transform} applies monotonic transformations to an integrand and a space or list structure of spaces. Common use cases include the probability integral transform, the transformation of infinite limits to finite ones and function dimensions to interval dimensions.

\subsection*{Usage}

\texttt{nint_transform(f, space, trans, funcDimToF = \emptyset, zeroInf = \emptyset)}

\subsection*{Arguments}

- \texttt{f} \quad function(x, \ldots), an integrand.
- \texttt{space} \quad a space or list structure of spaces.
- \texttt{trans} \quad a list of named lists, each containing \texttt{dIdcs}, \texttt{g} and \texttt{giDgi} or \texttt{giDg}, where
  - \texttt{dIdcs} is an integer vector of indices, the dimensions to transform
  - \texttt{g=function(x[dIdcs])} mapping \texttt{x[dIdcs]} to \texttt{y}
  - \texttt{giDgi=function(y)} returning a list of two, the inverse \texttt{gi(y) = x[dIdcs]}
    and the first derivatives of \texttt{gi(y)} with respect to \texttt{y}
  - or \texttt{giDg=function(y)} returning the inverse and the first derivatives of \texttt{g(x[dIdcs])}
    with respect to \texttt{x[dIdcs]}.
- \texttt{funcDimToF} \quad an integer vector of indices, the dimensions to look for function dimensions to transform to interval dimensions. \texttt{\emptyset} indicates all dimensions.
- \texttt{zeroInf} \quad a single value, used when \texttt{f} returns \texttt{\emptyset} and the Jacobian is infinite.

\subsection*{Details}

Interval dimensions and function dimensions returning interval dimensions only.

If a transformation is vector valued, that is \texttt{y = c(y1, \ldots, yn) = g(c(x1, \ldots, xn))}, then each component of \texttt{y} shall exclusively depend on the corresponding component of \texttt{x}. So \texttt{y[i] = g[i](x[i])} for an implicit function \texttt{g[i]}.

The transformation of function dimensions to interval dimensions is performed after the transformations defined by \texttt{trans}. Consecutive linear transformations, \texttt{g(x[dIdx]) = (x[dIdx] - d(x)[1])/(d(x)[2] - d(x)[1])} where \texttt{d} is the function dimension at dimension \texttt{dIdx}, are used. Deciding against this transformation probably leads to considerable loss in computational performance.
\textbf{Value}

\texttt{nint\_transform} returns either a named list containing the transformed integrand and space, or a list of such.

\textbf{See Also}

\texttt{nint\_integrate}, \texttt{nint\_space}, \texttt{nint\_tanTransform}, \texttt{fisherI}

\textbf{Examples}

\begin{verbatim}
library(mvtnorm)
library(SparseGrid)

dfltNCube = nint_integrateNCube

## 1D, normal pdf
mu = 137
sigma = mu/6
f = function(x) dnorm(x, mean=mu, sd=sigma)
space = nint_space(nint_intvDim(-Inf, Inf))

tt = nint_transform(f, space,
    list(nint_tanTransform(mu + 3, sigma*1.01, d1dcs=1)))
tt$space
ff = Vectorize(tt$f); curve(ff(x), tt$space[[1]][1], tt$space[[1]][2])
nint_integrate(tt$f, tt$space) # returns 1

## 2D, normal pdf

## prepare for SparseGrid
ncube = function(dimension)
    SparseGrid::createIntegrationGrid('GQU', dimension, 7) # rather sparse!
ncube = nint_integrateNCube_SparseGrid(ncube)
unlockBinding('nint_integrateNCube', environment(nint_integrate))
assign('nint_integrateNCube', ncube, envir=environment(nint_integrate))

mu = c(1, 2)
sigma = matrix(c(1, 0.7,
    0.7, 2), nrow=2)
f = function(x) {
    if (all(is.infinite(x))) # dmvnorm returns NaN in this case
        return(0)
    return(dmvnorm(x, mean=mu, sigma=sigma))
}

# plot f
x1 = seq(-1, 3, length.out=51); x2 = seq(-1, 5, length.out=51)
y = outer(x1, x2, function(x1, x2) apply(cbind(x1, x2), 1, f))
contour(x1, x2, y, xlab='x[1]', ylab='x[2]', main='f')
\end{verbatim}
```r
# nint_transform

space = nint_space(nint_intvDim(-Inf, Inf),
  nint_intvDim(-Inf, Inf))

tt = nint_transform(f, space,
  list(nint_tantransform(mu, diag(sigma), dIdcs=1:2)))

# plot tt$f
x1 = seq(tt$space[[1]][1], tt$space[[1]][2], length.out=51)
x2 = seq(tt$space[[2]][1], tt$space[[2]][2], length.out=51)
y = outer(x1, x2, function(x1, x2) apply(cbind(x1, x2), 1, tt$f))
contour(x1, x2, y, xlab='x[1]', ylab='x[2]', main='tt$f')

nint_integrate(tt$f, tt$space) # doesn't return 1
# tan transform is inaccurate here

# probability integral transform
dsigma = diag(sigma)
t1 = list(g=function(x) pnorm(x, mean=mu, sd=dsigma),
  giDg=function(y) {
    x = qnorm(y, mean=mu, sd=dsigma)
    list(x, dnorm(x, mean=mu, sd=dsigma))
  },
  dIdcs=1:2)

tt = nint_transform(f, space, list(t1))

# plot tt$f
x1 = seq(tt$space[[1]][1], tt$space[[1]][2], length.out=51)
x2 = seq(tt$space[[2]][1], tt$space[[2]][2], length.out=51)
y = outer(x1, x2, function(x1, x2) apply(cbind(x1, x2), 1, tt$f))
contour(x1, x2, y, xlab='x[1]', ylab='x[2]', main='tt$f')

nint_integrate(tt$f, tt$space) # returns almost 1

## 2D, half sphere
f = function(x) sqrt(1 - x[1]^2 - x[2]^2)
space = nint_space(nint_intvDim(~-1, 1),
  nint_funcDim(function(x)
    nint_intvDim(c(-1, 1)*sqrt(1 - x[1]^2))))

# plot f
x = seq(-1, 1, length.out=51)
y = outer(x, x, function(x1, x2) apply(cbind(x1, x2), 1, f))
persp(x, x, y, theta=45, phi=45, xlab='x[1]', ylab='x[2]', zlab='f')

tt = nint_transform(f, space, list())

# plot tt$f
x1 = seq(tt$space[[1]][1], tt$space[[1]][2], length.out=51)
```
\[ x_2 = \text{seq}(tt$space[[2]][1], tt$space[[2]][2], \text{length.out}=51) \]
\[ y = \text{outer}(x_1, x_2, \text{function}(x_1, x_2) \text{apply}(\text{cbind}(x_1, x_2), 1, tt$f)) \]
\[ \text{persp}(x_1, x_2, y, \theta=45, \phi=45, xlab='x[1]', ylab='x[2]', zlab='tt$f') \]

\text{nint\_integrate}(tt$f, tt$space) \# returns almost \(4/3\pi / 2\)

```r
## 2D, constrained normal pdf
f = \text{function}(x) \prod(\text{dnorm}(x, 0, 1))
space = \text{nint\_space}(\text{nint\_intvDim}(-\text{Inf}, \text{Inf}),
\quad \text{nint\_funcDim}(\text{function}(x) \text{nint\_intvDim}(-\text{Inf}, x[1]^2))))

\text{tt} = \text{nint\_transform}(f, space, \text{list}(\text{nint\_tanTransform}(0, 1, dIds=1:2)))

\# plot tt$f
x_1 = \text{seq}(tt$space[[1]][1], tt$space[[1]][2], \text{length.out}=51)
x_2 = \text{seq}(tt$space[[2]][1], tt$space[[2]][2], \text{length.out}=51)
y = \text{outer}(x_1, x_2, \text{function}(x_1, x_2) \text{apply}(\text{cbind}(x_1, x_2), 1, tt$f))
\text{persp}(x_1, x_2, y, \theta=45, \phi=45, xlab='x[1]', ylab='x[2]', zlab='tt$f')

\text{nint\_integrate}(tt$f, tt$space) \# Mathematica returns \(0.716315\)
```

\text{assign('nint\_integrateNCube', dfltNCube, envir=environment(nint\_integrate))}

---

### \text{nint\_TYPE} 

<table>
<thead>
<tr>
<th>Dimension Type</th>
<th>Attribute Values</th>
</tr>
</thead>
</table>

**Description**

A dimension object is identified by its dimension type attribute "\text{nint\_dtype}". On creation it is set to one of the following. See dimension types in "See Also" below.

**Usage**

\[ \text{nint\_TYPE\_SCAT\_DIM} \# = 1 \]
\[ \text{nint\_TYPE\_GRID\_DIM} \# = 2 \]
\[ \text{nint\_TYPE\_INTV\_DIM} \# = 3 \]
\[ \text{nint\_TYPE\_FUNC\_DIM} \# = 4 \]

**Format**

integer

**See Also**

\text{nint\_scatDim, nint\_gridDim, nint\_intvDim, nint\_funcDim, nint\_space}
nint_validateSpace  Validate Space

Description

nint_validateSpace performs a couple of checks on a space or list structure of spaces to ensure it is properly defined.

Usage

nint_validateSpace(x)

Arguments

x  a space or list structure of spaces.

Value

nint_validateSpace returns 0 if everything is fine, or an error code. See nint_ERROR.

See Also

nint_ERROR, nint_space

Examples

```python
## valid
s = nint_space()
s
nint_validateSpace(s)

s = nint_space(nint_intvDim(-1, 1))
s
nint_validateSpace(s)

## -1001
s = nint_space(1)
s
nint_validateSpace(s)

## -1002
s = nint_space(list(nint_scatDim(c(1, 2)), nint_scatDim(c(1, 2, 3)))))
s
nint_validateSpace(s)

s = nint_space(nint_scatDim(c(1, 2)),
                 nint_scatDim(c(1, 2, 3)))
s
nint_validateSpace(s)
```
**Build Derivative Function for Log f**

**Description**

`numDerivLogf` builds a function that evaluates to the first/second derivative of log(f(y, theta, ...)) with respect to theta[1]/theta[2] and theta[3].

**Usage**

```r
numDerivLogf(f, isLogf = FALSE, logZero = .Machine$double.xmin,
             logInf = .Machine$double.xmax/2, method = "Richardson",
             side = NULL, method.args = list())
```

**Arguments**

- `f` function(y, theta, ...), where theta is a list of parameters. A joint probability density function.
- `isLogf` set to TRUE if f is already log(f).
- `logZero` the value log(f) should return if f evaluates to 0.
- `logInf` the value log(f) should return if f evaluates to Inf.
- `method`, `side`, `method.args`

**Details**

`numDeriv` produces NaNs if the log evaluates to (negative) Inf so you may want to specify logZero and logInf.

`numDerivLogf` passes method, side and method.args directly to `numDeriv::grad`.

`numDeriv2Logf` duplicates the internals of `numDeriv::hessian` to gain speed. The defaults for method.args are list(eps=1e-4, d=0.1, zero.tol=sqrt(.Machine$double.eps/7e-7), r=4, v=2).
Value

numDerivLogf returns function(y, theta, i, ...) which evaluates to the first derivative of log(f(y, theta, ...)) with respect to theta[i].

numDeriv2Logf returns function(y, theta, i, j, ...) which evaluates to the second derivative of log(f(y, theta, ...)) with respect to theta[i] and theta[j].

See Also

grad and hessian in package numDeriv, buildf, DerivLogf, fisheri

Examples

## see examples for param

---

param     Parametric Model

Description

param creates an initial parametric model object. Unlike other model statements this function does not perform any computation.

Usage

param(fisherIf, dDim)

Arguments

fisherIf function(x, ...), where x is a vector, usually a point from the design space. It shall evaluate to the Fisher information matrix.

dDim length of x, usually the dimensionality of the design space.

Value

param returns an object of class "param". An object of class "param" is a list containing at least the following components:

- fisherIf: argument
- x: a row matrix of points where fisherIf has already been evaluated.
- fisherI: a list of Fisher information matrices, for each row in x respectively.

See Also

fisheri, update.param, Dsensitivity, getM, Defficiency
Examples

library(copula)

dfltNCube = nint_integrateNCube

## prepare for SparseGrid integration
ncube = function(dimension) {
    SparseGrid::createIntegrationGrid('GQU', dimension, 3)
}
ncube = nint_integrateNCube_SparseGrid(ncube)
unlockBinding('nint_integrateNCube', nint_integrate)
assign('nint_integrateNCube', ncube, envir=environment(nint_integrate))

## general settings
numDeriv = FALSE

## build pdf, derivatives
etas = function(theta) with(theta, {
    xx = x^c(0:4)
    c(c(beta1, beta2, beta3) %% xx[1], 2, 3), # x^c(0, 1, 2)
    c(beta4, beta5, beta6) %% xx[2:4, 5]) # x^c(1, 3, 4)
})
copula = claytonCopula()
alphas = c('alpha')
parNames = c(paste('beta', 1:6, sep=''), alphas)

if (numDeriv) {
    margins = function(y, theta, ...) {
        e = etas(theta)
        cbind(dnorm(y, mean=e, sd=1), pnorm(y, mean=e, sd=1))
    }
    f = buildf(margins, TRUE, copula, parNames=alphas)
    d2logf = numDeriv2Logf(f)
} else {
    es = list(
        eta1=quote(theta$beta1 + theta$beta2*theta$x + theta$beta3*theta$x^2),
        eta2=quote(theta$beta4*theta$x + theta$beta5*theta$x^3 + theta$beta6*theta$x^4))
    margins = list(list(pdf=substitute(dnorm(y[1], mean=eta1, sd=1), es),
                        cdf=substitute(pnorm(y[1], mean=eta1, sd=1), es)),
                  list(pdf=substitute(dnorm(y[2], mean=eta2, sd=1), es),
                        cdf=substitute(pnorm(y[2], mean=eta2, sd=1), es)))
    pn = as.list(alphas); names(pn) = alphas # map parameter to variable
    f = buildf(margins, TRUE, copula, parNames=pn)
cat('building derivatives ...')
tt = system.time(d2logf <- Deriv2Logf(f, parNames))
cat('
')
print(tt)
}

f
str(d2logf)

## param
model = function(theta) {
  integrand = function(y, theta, i, j)
  -d2logf(y, theta, i, j) * f(y, theta)
  yspace = nint_space(nint_intvDim(-Inf, Inf),
                      nint_intvDim(-Inf, Inf))

  fisherIf = function(x) {
    theta$x = x

    ## probability integral transform
    e = etas(theta)

    tt = nint_transform(integrand, yspace, list(list(
        dIdcs=1:2,
        g=function(y) pnorm(y, mean=e, sd=1),
        giDg=function(z) {
          t1 = qnorm(z, mean=e, sd=1)
          list(t1, dnorm(t1, mean=e, sd=1))
        }
    )))

    fisherI(tt$f, theta, parNames, tt$space)
  }

  return(param(fisherI, 1))
}

theta = list(beta1=1, beta2=1, beta3=1,
             beta4=1, beta5=1, beta6=1,
             alpha=ITau(copula, 0.5), x=0)

m = model(theta)

## update.param
system.time(m <- update(m, matrix(seq(0, 1, length.out=101), ncol=1)))

## find D-optimal design
D = Dsensitivity(defaults=list(x=m$x, desx=m$x, mod=m))

d <- Wynn(D, 7.0007, maxIter=1e4)
d$tag$Wynn$tolBreak
plot.desigh creates a one-dimensional design plot, optionally together with a specified sensitivity curve. If the design space has additional dimensions, the design is projected on a specified margin.
Usage

```r
## S3 method for class 'design'
plot(x, sensx = NULL, sens = NULL, sensTol = NULL,
    ..., margins = NULL, desSens = T, sensPch = "+",
sensArgs = list())
```

Arguments

- `x`: a design.
- `sensx`: (optional) a row matrix of points.
- `sens`: (optional) either a vector of sensitivities or a sensitivity function. The latter shall rely on defaults, see `dsensitivity` for details.
- `sensTol`: (optional) a single numeric. Adds a horizontal line at this sensitivity level.
- `...`: other arguments passed to `plot`.
- `margins`: a vector of indices, the dimensions to project on. Defaults to `1`.
- `desSens`: if `TRUE` and `sens` is not specified then the sensitivity function which potentially was used in `wynn` is taken as `sens`.
- `sensPch`: either a character vector of point 'characters' to add to the sensitivity curve or `NULL`.
- `sensArgs`: a list of arguments passed to draw calls related to the sensitivity.

References


See Also

design, dsensitivity

Examples

```r
## see examples for param
```

**Description**

`print.nint_space` prints a space in a convenient way.

Usage

```r
## S3 method for class 'nint_space'
print(x, ...)
```
reduce

Arguments

x a space.
... ignored.

Details

Each line represents a dimension. Format: "<dim idx>: <dim repr>". Each dimension has its own representation which should be easy to understand. nint_scatDim representations are marked by "s()".

See Also

nint_space

reduce Reduce Design

Description

reduce drops insignificant points and merges points in a certain neighbourhood.

Usage

reduce(des, distMax, wMin = 1e-06)

Arguments

des a design.
distMax maximum euclidean distance between points to be merged.
wMin minimum weight a point shall have to be considered significant.

Value

reduce returns an object of class "desigh". See design for its structural definition.

See Also

design

Examples

## see examples for param
**Description**

`rowmatch` returns a vector of the positions of (first) matches of the rows of its first argument in the rows of its second.

**Usage**

```r
rowmatch(x, table, nomatch = NA_integer_)
```

**Arguments**

- `x`: a row matrix of doubles, the rows to be matched.
- `table`: a row matrix of doubles, the rows to be matched against.
- `nomatch`: the value to be returned in the case when no match is found. Note that it is coerced to integer.

**Details**

`rowmatch` uses compiled C-code.

**Value**

`rowmatch` returns an integer vector giving the position of the matching row in `table` for each row in `x`. And `nomatch` if there is no matching row.

**See Also**

`match`

**Examples**

```r
a = as.matrix(expand.grid(as.double(2:3), as.double(3:6)))
a = a[sample(nrow(a)),]
a

b = as.matrix(expand.grid(as.double(3:4), as.double(2:5)))
b = b[sample(nrow(b)),]
b

i = rowmatch(a, b)
i
b[na.omit(i),] # matching rows
a[is.na(i),] # non matching rows
```
**roworder**  

*Matrix Ordering Permutation*

**Description**

roworder returns a permutation which rearranges the rows of its first argument into ascending order.

**Usage**

roworder(x, ...)

**Arguments**

x: a matrix.

...: other arguments passed to order.

**Value**

roworder returns an integer vector.

**See Also**

order

**Examples**

```r
x = expand.grid(1:3, 1:2, 3:1)
x = x[sample(seq(1, nrow(x)), nrow(x)),]
x
ord = roworder(x)
ord
x[ord,]
```

**rowsduplicated**  

*Determine Duplicate Rows*

**Description**

rowsduplicated determines which rows of a matrix are duplicates of rows with smaller subscripts, and returns a logical vector indicating which rows are duplicates.

**Usage**

rowsduplicated(x)
Arguments

\( x \)  
a row matrix of doubles.

Details

\texttt{rowsduplicated} uses compiled C-code.

Value

\texttt{rowsduplicated} returns a logical vector with one element for each row.

See Also

\texttt{duplicated}

\begin{verbatim}
seq1                       Sequence Generation
\end{verbatim}

Description

\texttt{seq1} is similar to \texttt{seq}, however by is strictly 1 by default and \texttt{integer(0)} is returned if the range is empty.

Usage

\texttt{seq1(from, to, by = 1)}

Arguments

\texttt{from, to, by}  \texttt{see seq}.

Value

\texttt{seq1} returns either \texttt{integer(0)} if range is empty or what an appropriate call to \texttt{seq} returns otherwise.

See examples below.

See Also

\texttt{seq}

Examples

\begin{verbatim}
seq1(1, 3)
seq1(3, 1) # different from seq
seq(3, 1)
3:1

seq1(5, 1, -3)
\end{verbatim}
**update.param**

*Update Parametric Model*

---

**Description**

`update.param` evaluates the Fisher information at uncharted points and returns an updated model object.

**Usage**

```r
## S3 method for class 'param'
update(object, x, ...)
```

**Arguments**

- `object` a model.
- `x` either a row matrix of points or a design, or a list structure of matrices or designs. The number of columns/the dimensionality of the design space shall be equal to `ncol(object$x)`.
- `...` ignored.

**Details**

When the user interrupts execution, the function returns a partially updated model object.

**Value**

`update.param` returns an object of class "param". See `param` for its structural definition.

**See Also**

`param, grow.grid, design`

**Examples**

```r
## see examples for param
```
**Description**

The `wdefficiency` function computes the weighted D-, D_s or D_A-efficiency measure for a design with respect to a reference design.

**Usage**

```
wdefficiency(des, ref, mods, modw, A = NULL, parNames = NULL)
```

**Arguments**

- `des` : a design.
- `ref` : a design, the reference.
- `mods` : a list of models.
- `modw` : a vector of weights.
- `A` : for
  - D-efficiency: NULL
  - D_s-efficiency: a vector of names or indices, the subset of parameters of interest.
  - D_A-efficiency: either
    - directly: a matrix without row names.
    - indirectly: a matrix with row names corresponding to the parameters.
- `parNames` : a vector of names or indices, the subset of parameters to use. Defaults to the parameters for which the Fisher information is available.

**Details**

Indices supplied to argument `A` correspond to the subset of parameters defined by argument `parNames`.

Weighted D efficiency is defined as

\[
\left( \frac{\exp \int_B \log |M(\xi, \bar{\theta})| dB}{\exp \int_B \log |M(\xi^*, \bar{\theta})| dB} \right)^{1/n}
\]

and weighted D_A efficiency as

\[
\left( \frac{\exp \int_B \log |A^T M(\xi, \bar{\theta})^{-1} A|^{-1} dB}{\exp \int_B \log |A^T M(\xi^*, \bar{\theta})^{-1} A|^{-1} dB} \right)^{1/s}
\]

**Value**

`wdefficiency` returns a single numeric.
wDsensitivity

See Also
design, param, Dsensitivity

wDsensitivity Weighted D Sensitivity

Description

wDsensitivity builds a sensitivity function for the weighted D-, D_s or D_A-optimality criterion which relies on defaults to speed up evaluation. Wynn for instance requires this behaviour/protocol.

Usage

wDsensitivity(A = NULL, parNames = NULL, defaults = list(x = NULL, desw = NULL, desx = NULL, mods = NULL, modw = NULL))

Arguments

A	for

• D-optimality: NULL
• D_s-optimality: a vector of names or indices, the subset of parameters of interest.
• D_A-optimality: either
  – directly: a matrix without row names.
  – indirectly: a matrix with row names corresponding to the parameters.
parNames	a vector of names or indices, the subset of parameters to use. Defaults to the parameters for which the Fisher information is available.
defaults	a named list of default values. The value NULL is equivalent to absence.

Details

Indices and rows of an unnamed matrix supplied to argument A correspond to the subset of parameters defined by argument parNames.

For efficiency reasons the returned function won’t complain about missing arguments immediately, leading to strange errors. Please ensure that all arguments are specified at all times. This behaviour might change in future releases.

Value

wDsensitivity returns function(x=NULL, desw=NULL, desx=NULL, mods=NULL, modw=NULL), the sensitivity function. It’s attributes contain this function’s arguments.

See Also
docopulae, param, Dsensitivity, Wynn, plot.desigh
Description

Wynn finds an optimal design using a sensitivity function and a Wynn-algorithm.

Usage

Wynn(sensF, tol, maxIter = 10000)

Arguments

- sensF: function(x=NULL, desw=NULL, desx=NULL, mod=NULL), a sensitivity function. Its attribute "defaults" shall contain identical x and desx, and sensF(desw=w) shall return sensitivities corresponding to each point in x.
- tol: the tolerance level regarding the sensitivities.
- maxIter: the maximum number of iterations.

Details

See Dsensitivity and its return value for a reference implementation of a function complying with the requirements for sensF.

The algorithm starts from a uniform weight design. In each iteration weight is redistributed to the point which has the highest sensitivity. Sequence: 1/i. The algorithm stops when all sensitivities are below a specified tolerance level or the maximum number of iterations is reached.

Value

Wynn returns an object of class "design". See design for its structural definition.

References


See Also

Dsensitivity, design

Examples

## see examples for param
Index

*Topic datasets
  nint_ERROR, 14
  nint_TYPE, 29

adaptIntegrate, 19

buildf, 3, 7, 11, 32

Cache, 3, 4
createIntegrationGrid, 19

Defficiency, 6, 8, 32, 43
Deriv, 7
Deriv2Logf (DerivLogf), 7
DerivLogf, 4, 7, 11, 32
design, 6, 8, 12, 36, 37, 41, 43, 44
docopulae, 9, 10, 43
docopulae-package (docopulae), 9
dSensitivity, 9, 9, 32, 36, 43, 44
duplicated, 40

fisheri, 4, 7, 11, 17, 24, 27, 32

getm, 8, 12, 32
grad, 31, 32
grow.grid, 12, 41

hessian, 31, 32

integrate, 13, 19
integrateA, 13, 19, 21

match, 38

nint_ERROR, 14, 30
nint_ERROR_DIM_TYPE (nint_ERROR), 14
nint_ERROR_SCATTER_LENGTH (nint_ERROR), 14
nint_ERROR_SPACE_DIM (nint_ERROR), 14
nint_ERROR_SPACE_TYPE (nint_ERROR), 14
nint_expandSpace, 14, 24

nint_funcDim, 15, 24, 29
nint_gridDim, 16, 24, 29
nint_integrate, 11, 16, 19, 21, 24, 27
nint_integrateNCube, 17, 18
nint_integrateNCube_cubature
  (nint_integrateNCube), 18
nint_integrateNCube_integrate
  (nint_integrateNCube), 18
nint_integrateNCube_SparseGrid
  (nint_integrateNCube), 18
nint_integrateNFunc, 17, 20
nint_integrateNFunc_recursive
  (nint_integrateNFunc), 20
nint_intvDim, 22, 24, 29
nint_scatDim, 23, 24, 29
nint_space, 11, 14–17, 23, 24, 27, 29, 30, 37
nint_tanTransform, 25, 27
nint_transform, 11, 17, 25, 26
nint_TYPE, 16, 23, 29
nint_TYPE_FUNC_DIM (nint_TYPE), 29
nint_TYPE_GRID_DIM (nint_TYPE), 29
nint_TYPE_INTV_DIM (nint_TYPE), 29
nint_TYPE_SCAT_DIM (nint_TYPE), 29
nint_validateSpace, 14, 24, 30
numDeriv2Logf (numDerivLogf), 31
numDerivLogf, 4, 7, 11, 31

order, 39

param, 6, 10–12, 32, 41, 43
plot.design, 8, 10, 35, 43
print.nint_space, 36

reduce, 8, 37
rowmatch, 38
roworder, 39
rowsduplicated, 39

seq, 40
seq1, 40
Simplify, 3, 4
update.param, 8, 13, 32, 41
wDefficiency, 6, 42
wDsensitivity, 10, 43
Wynn, 8, 10, 43, 44