Package ‘docopulae’

June 21, 2018

Title  Optimal Designs for Copula Models
Version  0.3.5
Date  2018-06-20
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  6.0.1
Repository  CRAN
Date/Publication  2018-06-21 13:59:29 UTC
**R topics documented:**

buildf .................................................. 2
Deficiency ........................................... 5
DerivLogf ............................................ 7
design .................................................. 8
docopulae ............................................. 8
Dsensitivity ........................................... 9
fisherI .................................................. 10
getM .................................................... 11
grow.grid ............................................... 12
integrateA ............................................. 12
nint_ERROR ............................................ 13
nint_expandSpace .................................... 14
nint_funcDim ......................................... 14
nint_gridDim .......................................... 15
nint_integrate ........................................ 16
nint_integrateNCube .................................. 18
nint_integrateNFunc .................................. 20
nint_intvDim ......................................... 22
nint_scatDim ......................................... 22
nint_space ........................................... 23
nint_tanTransform ................................... 24
nint_transform ....................................... 25
nint_TYPE ............................................ 28
nint_validateSpace ................................... 29
numDerivLogf ......................................... 30
param .................................................. 31
plot.desigh ........................................... 35
print.nint_space ...................................... 36
reduce ............................................... 36
rowmatch ............................................ 37
roworder ............................................. 38
seq1 .................................................. 39
update.param ......................................... 39
Wynn ................................................... 40

Index 42

---

**buildf**

*Build probability density or mass Function*

**Description**

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

\texttt{buildf(margins, continuous, copula, parNames = NULL, simplifyAndCache = T)}

Arguments

margins \( \) either

\begin{itemize}
  \item \texttt{function(y, theta, ...)}, where \texttt{theta} is a list of parameters. It shall return a column matrix of two, the probability densities and cumulative distributions.
  \item a list of pairs of expressions, each named "pdf" and "cdf", the probability density and cumulative distribution.
\end{itemize}

\texttt{continuous} \texttt{TRUE} if \texttt{margins} are continuous. See details.

copula \texttt{if} \texttt{margins} is

\begin{itemize}
  \item \texttt{a function} then either a copula object from package \texttt{copula} or \texttt{function(u, theta, ...)}, a probability density function if \texttt{continuous} else a cumulative distribution function.
  \item \texttt{a list} then either a copula object from package \texttt{copula} which contains distribution expressions or an expression for the probability density if \texttt{continuous} else the cumulative distribution which uses \texttt{u1,u2,}...
\end{itemize}

\texttt{parNames} \texttt{if} \texttt{margins} is a function and \texttt{copula} is a copula object then a vector of names or indices, the sequence of copula parameters in \texttt{theta}. \texttt{0} or \texttt{""} identifies copula parameters to skip.

\texttt{parNames} \texttt{if} \texttt{margins} is a list and \texttt{copula} is a copula object then a named list of names or indices, mapping parameters in \texttt{theta} to copula parameter variables. See \texttt{copula@exprdist}.

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

Details

Please note that expressions are not validated.

If \texttt{continuous} is \texttt{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

\texttt{buildf} returns \texttt{function(y, theta, ...)}, the joint probability density or mass function.

See Also

copula, \texttt{Simplify}, \texttt{Cache}, \texttt{numDerivLogf}, \texttt{DerivLogf}, \texttt{fisherI}
Examples

## for an actual use case see examples for param

```r
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(y1, theta$mu1, 1)),
                   cdf=quote(pnorm(y1, theta$mu1, 1))),
               list(pdf=quote(dnorm(y2, theta$mu2, 1)),
                   cdf=quote(pnorm(y2, theta$mu2, 1))))
copula = claytonCopula()

# args: list, copula object, parNames
f2 = buildf(margins, TRUE, copula, list(alpha='alpha'))
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))
```
Defficiency

**Description**

Defficiency computes the D-, D_s or D_A-efficiency measure for a design with respect to a reference design.
Usage

\[\text{Defficiency}(\text{des, ref, mod, } A = \text{NULL, parNames = NULL})\]

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

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

and D_A efficiency as

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

Value

Defficiency returns a single numeric.

See Also

design, param

Examples

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

Build Derivative Function for Log f

Description

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

Usage

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

DerivRLogf(f, parNames, preSimplify = T, ...)

Arguments

- \( f \) function(y, theta, ...), 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 DerivRLogf.

Value

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

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

See Also

Deriv, Deriv in package Deriv, buildf, numDerivLogf, fisherI

Examples

```r
## see examples for param
## mind the gain regarding runtime compared to numDeriv
```
**design**  

*Design*

**Description**

design creates a custom design object.

**Usage**

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 1.

- **tag**  
a list containing additional information about the design.

**Value**

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

**See Also**

Wynn, reduce, getM, plot.desigh, Defficiency, 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

Dsensitivity

Description

Dsensitivity 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

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

`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, Wynn, plot.desigh

Examples

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

---

### fisherI

**Fisher Information**

**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]]`. 
getM

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
## 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**

```r
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.

**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**

```r
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`.  

nint_ERROR

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_ERROR  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
nint_funcDim

### nint_expandSpace  
**Expand Space**

**Description**

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

**Usage**

*nint_expandSpace*(x)

**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**

```plaintext
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 \texttt{nint_intvDim} or a vector.

Details

Obviously if \( x \) returns an object of type \texttt{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

\texttt{nint_scatDim} returns its argument with the dimension type attribute set to \texttt{nint_TYPE_FUNC_DIM}.

See Also

\texttt{nint_TYPE}, \texttt{nint_space}
**Description**

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

**Usage**

*nint_integrate*(f, 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

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

## probability integral transform

tt = nint_transform(function(x) prod(dnorm(x)), s, list(list(
dIdcs=1:2,
g=function(x) pnorm(x),
   giDg=function(y) { t1 = qnorm(y); list(t1, dnorm(t1)) })))

nint_integrate(tt$f, tt$s) # 1

## functionally dependent
## f) area of triangle

s = nint_space(nint_intvDim(0, 1),
               nint_funcDim(function(x) nint_intvDim(x[1]/2, 1 - x[1]/2)))
s
nint_integrate(function(x) 1, s) # 0.5

## g) area of circle

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

## h) volume of sphere

s = nint_space(s[[1]],
               s[[2]],
               nint_funcDim(function(x) {
       r = sin(acos(x[1]))
       nint_intvDim(c(-1, 1) * r*cos(asin(x[2] / r)))
   }) )
s
nint_integrate(function(x) 1, s) # 4*pi/3
nint_integrateNCube  

Integrate Hypercube

Description

Interface to the integration over interval dimensions.

Usage

nint_integrateNCube(f, lowerLimit, upperLimit, ...)

nint_integrateNCube_integrate(integrate)

nint_integrateNCube_cubature(adaptIntegrate)

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 ((subdivisions * 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)))
```
nint_integrateNFunc

Integrate N Function

Description

Interface to the integration over function dimensions.

Usage

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

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

## prepare for custom recursive implementation
using = TRUE
nfunc = nint_integrateNFunc_recursive(
    function(f, lowerLimit, upperLimit, ...) {
        if (using) { # this function is called many times
            using <<- FALSE
            cat('using integrateA\n')
        }
        integrateA(f, lowerLimit, upperLimit, ..., subdivisions=1)$value
    }
)
unlockBinding('nint_integrateNFunc', environment(nint_integrate))
assign('nint_integrateNFunc', nfunc, envir=environment(nint_integrate))

## integrate with custom recursive implementation
nint_integrate(function(x) 1, s) # pi

## prepare for custom solution
f = function(f, funcs, x0, i0, ...) {
    # add sophisticated code here
    print(list(f=f, funcs=funcs, x0=x0, i0=i0, ...))
    stop('do something')
}
unlockBinding('nint_integrateNFunc', environment(nint_integrate))
assign('nint_integrateNFunc', f, envir=environment(nint_integrate))

## integrate with custom solution
try(nint_integrate(function(x) 1, s))
nint_intvDim  Interval Dimension

Description

nint_intvDim 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

nint_intvDim returns a vector of length 2 with the dimension type attribute set to nint_TYPE_INTV_DIM.

See Also

nint_TYPE, nint_space

nint_scatDim  Scatter Dimension

Description

nint_scatDim is defined by a sequence of values. Together with other scatter dimensions it defines a sparse grid.

Usage

nint_scatDim(x)

Arguments

x  a vector of any type.
Details

Imagine using `cbind` to create a row matrix of points.

Value

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

See Also

`nint_TYPE`, `nint_space`

---

**nint_space**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>nint_space</code> defines an n-dimensional space as a list of dimensions. A space may consist of subspaces. A space without subspaces is called a true subspace.</td>
</tr>
</tbody>
</table>

**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)))
    )
```

nint_tanTransform  
Tangent Transform

Description

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

Usage

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

Arguments

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

Value

nint_tanTransform returns a named list of two functions "g" and "g_inv" 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, dIds=1)))
tt$space
ff = Vectorize(tt$f); curve(ff(x), tt$space[[1]][1], tt$space[[1]][2])

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

# same with larger mu
**nint_transform**  
*Transform Integral*

**Description**

*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.

**Usage**

```r
nint_transform(f, space, trans, funcDimToF = 0, zeroInf = 0)
```

**Arguments**

- **f**: `function(x, ...)`, an integrand.
- **space**: a space or list structure of spaces.
- **trans**: a list of named lists, each containing `dIdcs`, `g` and `gidgi` or `gidg`, where
  - `dIdcs` is an integer vector of indices, the dimensions to transform
  - `g = function(x[dIdcs])` mapping `x[dIdcs]` to `y`
  - `gidgi = function(y) returning a list of two, the inverse gi(y) = x[dIdcs] and the first derivatives of gi(y) with respect to y`
  - or `gidg = function(y) returning the inverse and the first derivatives of g(x[dIdcs]) with respect to x[dIdcs]`.
- **funcDimToF**: an integer vector of indices, the dimensions to look for function dimensions to transform to interval dimensions. 0 indicates all dimensions.
- **zeroInf**: a single value, used when `f` returns 0 and the Jacobian is infinite.
Details

Interval dimensions and function dimensions returning interval dimensions only.

If a transformation is vector valued, that is \( y = c(y_1, \ldots, y_n) = g(c(x_1, \ldots, x_n)) \), then each component of \( y \) shall exclusively depend on the corresponding component of \( x \). So \( y[i] = g[i](x[i]) \) for an implicit function \( g[i] \).

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

Value

\( \text{nint_transform} \) returns either a named list containing the transformed integrand and space, or a list of such.

See Also

\( \text{nint_integrate, nint_space, nint_tanTransform, fisheri} \)

Examples

```r
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, dIdxs=1)))

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

## 2D, normal pdf

## prepare for SparseGrid
cube = function(dimension)
    SparseGrid::createIntegrationGrid('GQU', dimension, 7) # rather sparse!
cube = nint_integrateNCube_SparseGrid(cube)
unlockBinding('nint_integrateNCube', environment(nint_integrate))
assign('nint_integrateNCube', cube, envir=environment(nint_integrate))
```
\[ \mu = c(1, 2) \]
\[ \sigma = \text{matrix}(c(1, 0.7, 0.7, 2), \text{nrow}=2) \]
\[ f = \text{function}(x) \{ \]
\[ \quad \text{if (all(is.infinite(x))) \# \text{dmvnorm returns NaN in this case} \]
\[ \quad \text{return(0)} \]
\[ \quad \text{return(dmvnorm(x, mean=mu, sigma=sigma))} \]
\[ \} \]
\[ \# \text{plot f} \]
\[ x1 = \text{seq}(-1, 3, \text{length.out}=51); x2 = \text{seq}(-1, 5, \text{length.out}=51) \]
\[ y = \text{outer}(x1, x2, \text{function}(x1, x2) \text{apply}(\text{cbind}(x1, x2), 1, f)) \]
\[ \text{contour}(x1, x2, y, \text{xlab}='x[1]', \text{ylab}='x[2]', \text{main}='f') \]
\[ \text{space} = \text{nint_space}((-\infty, \infty)) \]
\[ \text{nint_intvDim}(-\infty, \infty)) \]
\[ \text{tt} = \text{nint_transform}(f, \text{space}, \text{list}(\text{nint_tanTransform}((1, 2)))) \]
\[ \text{tt} \]
\[ \# \text{plot tt}\$f \]
\[ x1 = \text{seq}((\text{tt} \text{space})[[1]][1], \text{tt} \text{space}[[1]][2], \text{length.out}=51) \]
\[ x2 = \text{seq}((\text{tt} \text{space})[[2]][1], \text{tt} \text{space}[[2]][2], \text{length.out}=51) \]
\[ y = \text{outer}(x1, x2, \text{function}(x1, x2) \text{apply}(\text{cbind}(x1, x2), 1, \text{tt}\$f)) \]
\[ \text{contour}(x1, x2, y, \text{xlab}='x[1]', \text{ylab}='x[2]', \text{main}='\text{tt}\$f') \]
\[ \text{nint_integrate}((\text{tt}\$f, \text{tt} \text{space}) \# \text{doesn’t return 1} \]
\[ \# \text{tan transform is inaccurate here} \]
\[ \# \text{probability integral transform} \]
\[ \text{dsigma} = \text{diag}(\sigma) \]
\[ t1 = \text{list}(\text{g=function}(x) \text{pnorm}(x, \text{mean}=\mu, \text{sd}=\text{dsigma}), \]
\[ \text{giDg=function}(y) \{ \]
\[ \quad x = \text{qnorm}(y, \text{mean}=\mu, \text{sd}=\text{dsigma}) \]
\[ \quad \text{list}(x, \text{dnorm}(x, \text{mean}=\mu, \text{sd}=\text{dsigma})) \]
\[ \} \]
\[ \text{d1dc}=1:2) \]
\[ \text{tt} = \text{nint_transform}(f, \text{space}, \text{list}(t1)) \]
\[ \# \text{plot tt}\$f \]
\[ x1 = \text{seq}((\text{tt} \text{space})[[1]][1], \text{tt} \text{space}[[1]][2], \text{length.out}=51) \]
\[ x2 = \text{seq}((\text{tt} \text{space})[[2]][1], \text{tt} \text{space}[[2]][2], \text{length.out}=51) \]
\[ y = \text{outer}(x1, x2, \text{function}(x1, x2) \text{apply}(\text{cbind}(x1, x2), 1, \text{tt}\$f)) \]
\[ \text{contour}(x1, x2, y, \text{xlab}='x[1]', \text{ylab}='x[2]', \text{main}='\text{tt}\$f') \]
\[ \text{nint_integrate}((\text{tt}\$f, \text{tt} \text{space}) \# \text{returns almost 1} \]

\[ \text{## 2D, half sphere} \]
\[ f = \text{function}(x) \sqrt{1 - x[1]^2 - x[2]^2} \]
\[ \text{space} = \text{nint_space}((-1, 1)) \]
### nint_TYPE

#### Dimension Type Attribute Values

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A dimension object is identified by its dimension type attribute &quot;nint_dtype&quot;. On creation it is set to one of the following. See dimension types in &quot;See Also&quot; below.</td>
</tr>
</tbody>
</table>

#### Usage

```
nint_TYPE_SCAT_DIM # = 1
```
**nint_validateSpace**

nint_TYPE_GRID_DIM # = 2
nint_TYPE_INTV_DIM # = 3
nint_TYPE_FUNC_DIM # = 4

**Format**
integer

**See Also**
- 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**
```
## valid
s = nint_space()
s
nint_validateSpace(s)

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

**Build Derivative Function for Log f**

**Description**

numDerivLogf/numDeriv2Logf builds a function that evaluates to the first/second derivative of log(f(y, theta, ...)) with respect to theta[i]/theta[j] and theta[j].

**Usage**

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

numDeriv2Logf(f, isLogf = FALSE, logZero = .Machine$double.xmin, logInf = .Machine$double.xmax/2, method = "Richardson", 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
            see grad and hessian in package numDeriv.

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 deriva-
tive 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', environment(nint_integrate))
assign('nint_integrateNCube', ncube, envir=environment(nint_integrate))

## general settings
numDeriv = FALSE

## build pdf, derivatives
etas = function(theta) with(theta, {
  xx = x^'0:4
  c(c(beta1, beta2, beta3) %*% xx[c(1, 2, 3)], y x^'0, 1, 2
  c(beta4, beta5, beta6) %*% xx[c(2, 4, 5)]) %*% x^'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('n')
  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) {
        tl = qnorm(z, mean=e, sd=1)
        list(tl, dnorm(tl, mean=e, sd=1))
      })))

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

  return(param(fisherIf, 1))
}
theta = list(betalpha1=1, beta2=1, beta3=1,
               betalpha4=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$t$Wyn$tolBreak

dev.new(); plot(d, sens Tol=7, main='d')

getM(m, d)

rd = reduce(d, 0.05)
cbind(x=rd$x, w=rd$w)

dev.new(); plot(rd, main='rd')

try(getM(m, rd))
m2 = update(m, rd)
getM(m2, rd)

## find Ds-optimal design
s = c(alphas, 'betalpha1', 'betalpha2', 'betalpha3')
Ds = Dsensitivity(A=s, defaults=list(x=m$x, desx=m$x, mod=m))

ds <- Wynn(Ds, 4.0004, maxIter=1e4)
d$t$Wyn$tolBreak

dev.new(); plot(reduce(ds, 0.05), sens Tol=4, main='ds')

## create custom design
n = 4
d2 = design(x=matrix(seq(0, 1, length.out=n), ncol=1), w=rep(1/n, n))

m = update(m, d2)
dev.new(); plot(d2, sensx=d$x, sens=DS(x=d$x, desx=d2$x, desw=d2$w, mod=m),
                sens Tol=7, main='d2')

## compare designs
Defficiency(ds, d, m)
Defficiency(d, ds, m, A=s) # Ds-efficiency
Defficiency(d2, d, m)
Defficiency(d2, ds, m) # D-efficiency

## end with nice plot
dev.new(); plot(rd, main='rd')
plot.desigh

Description

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

## 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

## see examples for param
print.nint_space  

Print Space

Description

print.nint_space prints a space in a convenient way.

Usage

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

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

```R
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 "design". 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

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
```

Description

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

Usage

```r
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,]
```
Description
seq1 is similar to seq, however by is strictly 1 by default and integer(∅) is returned if the range is empty.

Usage
seq1(from, to, by = 1)

Arguments
from, to, by see seq.

Value
seq1 returns either integer(∅) if range is empty or what an appropriate call to seq returns otherwise.
See examples below.

See Also
seq

Examples
seq1(1, 3)
seq1(3, 1) # different from seq
seq(3, 1)
3:1

seq1(5, 1, -3)

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

Usage
## 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 \( \text{ncol} \( \text{object} \$ x \) \).

\( \ldots \) 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

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

Usage

Wynn(sensF, tol, maxIter = 10000)

Arguments

sensF function(\( x = \text{NULL} \), desw=\text{NULL}, desx=\text{NULL}, mod=\text{NULL}), a sensitivity function. It's attribute "defaults" shall contain identical \( x \) and \( \text{desx} \), and \( \text{sensF} (\text{desw} = \text{w}) \) 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

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

*Topic datasets
  nint_ERROR, 13
  nint_TYPE, 28

adaptIntegrate, 19

buildf, 2, 7, 11, 31

Cache, 3
createIntegrationGrid, 19

Defficiency, 5, 8, 32
Deriv, 7
Deriv2Logf (DerivLogf), 7
DerivLogf, 3, 7, 11, 31
design, 6, 8, 11, 35, 37, 40, 41
docopulae, 8, 10
docopulae-package (docopulae), 8
dsensitivity, 9, 9, 32, 35, 41

fisheri, 3, 7, 10, 16, 23, 26, 31, 32

getM, 8, 11, 32
grad, 31
grow.grid, 12, 40

hessian, 31

integrate, 12, 13, 19
integrateA, 12, 19, 21

match, 37

nint_ERROR, 13, 29
nint_ERROR_DIM_TYPE (nint_ERROR), 13
nint_ERROR_SCATTER_LENGTH (nint_ERROR), 13
nint_ERROR_SPACE_DIM (nint_ERROR), 13
nint_ERROR_SPACE_TYPE (nint_ERROR), 13
nint_expandspace, 14, 23
nint_funcDim, 14, 23, 29

nint_gridDim, 15, 23, 29
nint_integrate, 11, 16, 19, 21, 23, 26
nint_integrateNCube, 16, 18
nint_integrateNCube_cubature (nint_integrateNCube), 18
nint_integrateNCube_integrate (nint_integrateNCube), 18
nint_integrateNCube_SparseGrid (nint_integrateNCube), 18
nint_integrateNFunc, 16, 20
nint_integrateNFunc_recursive (nint_integrateNFunc), 20
nint_intvDim, 22, 23, 29
nint_scatDim, 22, 23, 29
nint_space, 11, 13–16, 22, 23, 26, 29, 36
nint_tanTransform, 24, 26
nint_transform, 11, 16, 24, 25
nint_TYPE, 15, 22, 23, 28
nint_TYPE_FUNC_DIM (nint_TYPE), 28
nint_TYPE_GRID_DIM (nint_TYPE), 28
nint_TYPE_INTV_DIM (nint_TYPE), 28
nint_validateSpace, 13, 23, 29
numDeriv2Logf (numDerivLogf), 30
numDerivLogf, 3, 7, 11, 30

order, 38

param, 6, 10, 11, 31, 40
plot.design, 8, 10, 35
print.nint_space, 36

reduce, 8, 36
rowmatch, 37
roworder, 38

seq, 39
seqL, 39
Simplify, 3

update.param, 8, 12, 32, 39
INDEX

Wynn, 8, 10, 40