

# Package ‘docopulae’

June 19, 2018

**Title** Optimal Designs for Copula Models

**Version** 0.3.4

**Date** 2018-06-18

**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** 2018-06-19 14:53:11 UTC

**R topics documented:**

buildf	2
Defficiency	5
DerivLogf	7
design	8
docopulae	8
Dsensitivity	9
fisherI	10
getM	11
integrateA	12
nint_ERROR	12
nint_expandSpace	13
nint_funcDim	14
nint_gridDim	15
nint_integrate	15
nint_integrateNCube	17
nint_integrateNFunc	19
nint_intvDim	21
nint_scatDim	22
nint_space	22
nint_tanTransform	23
nint_transform	24
nint_TYPE	28
nint_validateSpace	28
numDerivLogf	29
param	31
plot.design	34
print.nint_space	35
reduce	36
rowmatch	36
roworder	37
seq1	38
update.param	39
Wynn	40
<b>Index</b>	<b>41</b>

---

buildf	<i>Build probability density or mass Function</i>
--------	---

---

**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 <ul style="list-style-type: none"> <li>• <code>function(y, theta, ...)</code>, where <code>theta</code> is a list of parameters. It shall return a column matrix of two, the probability densities and cumulative distributions.</li> <li>• a list of pairs of expressions, each named "pdf" and "cdf", the probability density and cumulative distribution.</li> </ul>
continuous	TRUE if margins are continuous. See details.
copula	if margins is <ul style="list-style-type: none"> <li>• a function then either a copula object from package <b>copula</b> or <code>function(u, theta, ...)</code>, a probability density function if <code>continuous</code> else a cumulative distribution function.</li> <li>• a list then either a copula object from package <b>copula</b> which contains distribution expressions or an expression for the probability density if <code>continuous</code> else the cumulative distribution which uses <code>u1,u2,...</code></li> </ul>
parNames	if <ul style="list-style-type: none"> <li>• (optional) margins is a function and copula is a copula object then a vector of names or indices, the sequence of copula parameters in <code>theta</code>. <code>0</code> or "" identifies copula parameters to skip.</li> <li>• margins is a list and copula is a copula object then a named list of names or indices, mapping parameters in <code>theta</code> to copula parameter variables. See <code>copula@exprdist</code>.</li> </ul>
simplifyAndCache	(if margins is a list) simplify and cache the result using <a href="#">Simplify</a> and <a href="#">Cache</a> from package <b>Deriv</b> 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**

```

## 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='alpha1')
f1 # uses theta[['alpha1']] as copula parameter

## evaluate and plot
theta = list(mu1=2, mu2=-3, alpha1=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$alpha1
v = outer(y1, y2, function(yy1, yy2)
  dmvnorm(cbind(yy1, yy2), mean=c(theta$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()

# 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

*D Efficiency*


---

### Description

Defficiency computes the D-, D<sub>s</sub> or D<sub>A</sub>-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 <ul style="list-style-type: none"> <li>• D-efficiency: NULL</li> <li>• D_s-efficiency: a vector of names or indices, the subset of parameters of interest.</li> <li>• D_A-efficiency: either           <ul style="list-style-type: none"> <li>– directly: a matrix without row names.</li> <li>– indirectly: a matrix with row names corresponding to the parameters.</li> </ul> </li> </ul>
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

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

and D\_A efficiency as

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

**Value**

Defficiency returns a single numeric.

**See Also**

[design, param](#)

**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  $\log(f(y, \theta, \dots))$  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, ...), where theta is a list of parameters.
parNames	a vector of names or indices, the subset of parameters to use.
preSimplify	simplify the body of f using functions from package <b>Deriv</b> .
...	other arguments passed to <a href="#">Deriv</a> from package <b>Deriv</b> .

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

Deriv2Logf returns function(y, theta, i, j, ...) which evaluates to the second derivative of  $\log(f(y, \theta, \dots))$  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**

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

---

design	<i>Design</i>
--------	---------------

---

### 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 "design". An object of class "design" is a list containing at least this function's arguments.

### See Also

[Wynn](#), [reduce](#), [getM](#), [plot.design](#), [Defficiency](#), [update.param](#)

### Examples

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

---

docopulae	<i>Design of Experiments with Copulas</i>
-----------	---

---

### 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

E. Perrone & W.G. Müller (2016) Optimal designs for copula models, *Statistics*, 50:4, 917-929, DOI: 10.1080/02331888.2015.1111892

## See Also

[Dsensitivity](#)

---

Dsensitivity

*D Sensitivity*

---

## 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
	<ul style="list-style-type: none"> <li>• D-optimality: NULL</li> <li>• D_s-optimality: a vector of names or indices, the subset of parameters of interest.</li> <li>• D_A-optimality: either           <ul style="list-style-type: none"> <li>– directly: a matrix without row names.</li> <li>– indirectly: a matrix with row names corresponding to the parameters.</li> </ul> </li> </ul>
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**

E. Perrone & W.G. Müller (2016) Optimal designs for copula models, *Statistics*, 50:4, 917-929, DOI: 10.1080/02331888.2015.1111892

**See Also**

[docopulae](#), [param](#), [Wynn](#), [plot.design](#)

**Examples**

```
## 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  $d\log f/d2\log f$  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**

```
## 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
```

---

 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\_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_expandSpace	<i>Expand Space</i>
------------------	---------------------

---

**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**

```

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\_scadDim returns its argument with the dimension type attribute set to nint\_TYPE\_FUNC\_DIM.

**See Also**

[nint\\_TYPE](#), [nint\\_space](#)

---

nint_gridDim	<i>Grid Dimension</i>
--------------	-----------------------

---

**Description**

nint\_gridDim is defined by a sequence of values. Together with other grid dimensions it defines a dense grid.

**Usage**

```
nint_gridDim(x)
```

**Arguments**

x                    a vector of any type.

**Details**

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

**Value**

nint\_scattDim returns its argument with the dimension type attribute set to `nint_TYPE_GRID_DIM`.

**See Also**

[nint\\_TYPE](#), [nint\\_space](#)

---

nint_integrate	<i>Integrate</i>
----------------	------------------

---

**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**

```
## 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

## continous
## 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(
  dIds=1:2,
  g=function(x) pnorm(x),
  giDg=function(y) { t1 = qnorm(y); list(t1, dnorm(t1)) })))
```

```

tt$space
nint_integrate(tt$f, tt$space) # 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 ( $(\text{subdivisions} * 21) ** D$  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

```
## 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
ncube = function(dimension) {
  cat('using SparseGrid\n')
  SparseGrid::createIntegrationGrid('GQU', dimension, 7)
}
ncube = 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))

```

---

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))

assign('nint_integrateNFunc', dfltNFunc, envir=environment(nint_integrate))

```

---

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	<i>Scatter Dimension</i>
--------------	--------------------------

---

**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	<i>Space</i>
------------	--------------

---

**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**

```
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
```

---

nint_tanTransform	<i>Tangent Transform</i>
-------------------	--------------------------

---

**Description**

nint\_tanTransform creates the transformation  $g(x) = \text{atan}((x - \text{center})/\text{scale})$  to be used in nint\_transform.

**Usage**

```
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**

```

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

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

```

---

`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**

```
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 dIds, g and giDgi or giDg, where <ul style="list-style-type: none"> <li>• dIds is an integer vector of indices, the dimensions to transform</li> <li>• g=function(x[dIds]) mapping x[dIds] to y</li> <li>• giDgi=function(y) returning a list of two, the inverse <math>g_i(y) = x[dIds]</math> and the first derivatives of <math>g_i(y)</math> with respect to y</li> <li>• or giDg=function(y) returning the inverse and the first derivatives of <math>g(x[dIds])</math> with respect to x[dIds].</li> </ul>
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  $y = c(y_1, \dots, y_n) = g(c(x_1, \dots, x_n))$ , 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 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**

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

**See Also**

[nint\\_integrate](#), [nint\\_space](#), [nint\\_tanTransform](#), [fisherI](#)

**Examples**

```
library(mvtnorm)
library(SparseGrid)

df1tNCube = 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, 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) # 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')

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

tt = nint_transform(f, space,
                   list(nint_tanTransform(mu, diag(sigma), dIds=1:2)))
tt$space

# 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))
    },
    dIds=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())
tt$space

# 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))
persp(x1, x2, y, theta=45, phi=45, xlab='x[1]', ylab='x[2]', zlab='tt$f')

nint_integrate(tt$f, tt$space) # returns almost 4/3*pi / 2

## 2D, constrained normal pdf
f = function(x) prod(dnorm(x, 0, 1))
space = nint_space(nint_intvDim(-Inf, Inf),
                  nint_funcDim(function(x) nint_intvDim(-Inf, x[1]^2)))

tt = nint_transform(f, space, list(nint_tanTransform(0, 1, dIds=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))
persp(x1, x2, y, theta=45, phi=45, xlab='x[1]', ylab='x[2]', zlab='tt$f')

nint_integrate(tt$f, tt$space) # Mathematica returns 0.716315

```

```
assign('nint_integrateNCube', dfltNCube, envir=environment(nint_integrate))
```

---

nint_TYPE	<i>Dimension Type Attribute Values</i>
-----------	--

---

### Description

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

### Usage

```
nint_TYPE_SCAT_DIM # = 1
```

```
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	<i>Validate Space</i>
--------------------	-----------------------

---

### 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)

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

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

s = nint_space(nint_scatterDim(c(1, 2)),
              nint_scatterDim(c(1, 2, 3)))
s
nint_validateSpace(s)

## -1003
nint_validateSpace(1)
nint_validateSpace(list(nint_space())) # valid
nint_validateSpace(list(1))

## -1004
s1 = nint_space(nint_gridDim(1:3),
               nint_scatterDim(c(0, 1)))
s2 = nint_space(s1[[1]])
s1 # 2D
s2 # 1D
nint_validateSpace(list(s1, s2))
```

**Description**

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

numDeriv2Logf returns function( $y, \theta, i, j, \dots$ ) which evaluates to the second derivative of  $\log(f(y, \theta, \dots))$  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	<i>Parametric Model</i>
-------	-------------------------

---

### 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)], # x^c(0, 1, 2)
    c(beta4, beta5, beta6) %% xx[c(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('\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(
    dIds=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(fisherIf, 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

dev.new(); plot(d, sensTol=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, 'beta1', 'beta2', 'beta3')
Ds = Dsensitivity(A=s, defaults=list(x=m$x, desx=m$x, mod=m))

```

```

ds <- Wynn(Ds, 4.0004, maxIter=1e4)
ds$tag$Wynn$tolBreak

dev.new(); plot(reduce(ds, 0.05), sensTol=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$x, sens=D(x=d$x, desx=d2$x, desw=d2$w, mod=m),
               sensTol=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')

assign('nint_integrateNCube', dfltNCube, envir=environment(nint_integrate))

```

---

plot.design

*Plot Design*


---

### Description

plot.design 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 <a href="#">Dsensitivity</a> 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

uses add.alpha from <http://www.magesblog.com/2013/04/how-to-change-alpha-value-of-colours-in.html>

## 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

```
## 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\_scatterDim 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 "design". See [design](#) for its structural definition.

**See Also**

[design](#)

**Examples**

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

---

rowmatch

*Row Matching*

---

**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**

```
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**

```
x = expand.grid(1:3, 1:2, 3:1)
x = x[sample(seq1(1, nrow(x)), nrow(x)),]
x

ord = roworder(x)
ord

x[ord,]
```

---

 seq1

*Sequence Generation*


---

**Description**

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

**Usage**

```
seq1(from, to, by = 1)
```

**Arguments**

`from`, `to`, `by` see [seq](#).

**Value**

`seq1` returns either `integer(0)` 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)
```

---

update.param

*Update Parametric Model*


---

**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 <code>ncol(object\$x)</code> .
...	ignored.

**Value**

update.param returns an object of class "param". See [param](#) for its structural definition.

**See Also**

[param](#), [design](#)

**Examples**

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

---

 Wynn

 Wynn
 

---

### 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. It's 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 it's 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

Wynn, Henry P. (1970) The Sequential Generation of D-Optimum Experimental Designs. *The Annals of Mathematical Statistics*, 41(5):1655-1664.

### See Also

[Dsensitivity](#), [design](#)

### Examples

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

# Index

## \*Topic **datasets**

- nint\_ERROR, 12
- nint\_TYPE, 28
- adaptIntegrate, 18
- buildf, 2, 7, 11, 30
- Cache, 3
- createIntegrationGrid, 18
- Defficiency, 5, 8, 31
- Deriv, 7
- Deriv2Logf (DerivLogf), 7
- DerivLogf, 3, 7, 11, 30
- design, 6, 8, 11, 35, 36, 39, 40
- docopulae, 8, 10
- docopulae-package (docopulae), 8
- Dsensitivity, 9, 9, 31, 34, 35, 40
- fisherI, 3, 7, 10, 16, 23, 25, 30, 31
- getM, 8, 11, 31
- grad, 30
- hessian, 30
- integrate, 12, 18
- integrateA, 12, 18, 20
- match, 37
- nint\_ERROR, 12, 29
- nint\_ERROR\_DIM\_TYPE (nint\_ERROR), 12
- nint\_ERROR\_SCATTER\_LENGTH (nint\_ERROR), 12
- nint\_ERROR\_SPACE\_DIM (nint\_ERROR), 12
- nint\_ERROR\_SPACE\_TYPE (nint\_ERROR), 12
- nint\_expandSpace, 13, 23
- nint\_funcDim, 14, 23, 28
- nint\_gridDim, 15, 23, 28
- nint\_integrate, 11, 15, 18, 20, 23, 25
- nint\_integrateNCube, 16, 17
- nint\_integrateNCube\_cubature (nint\_integrateNCube), 17
- nint\_integrateNCube\_integrate (nint\_integrateNCube), 17
- nint\_integrateNCube\_SparseGrid (nint\_integrateNCube), 17
- nint\_integrateNFunc, 16, 19
- nint\_integrateNFunc\_recursive (nint\_integrateNFunc), 19
- nint\_intvDim, 21, 23, 28
- nint\_scatDim, 22, 23, 28
- nint\_space, 11, 13–16, 22, 22, 25, 28, 29, 35
- nint\_tanTransform, 23, 25
- nint\_transform, 11, 16, 24, 24
- nint\_TYPE, 14, 15, 22, 28
- nint\_TYPE\_FUNC\_DIM (nint\_TYPE), 28
- nint\_TYPE\_GRID\_DIM (nint\_TYPE), 28
- nint\_TYPE\_INTV\_DIM (nint\_TYPE), 28
- nint\_TYPE\_SCAT\_DIM (nint\_TYPE), 28
- nint\_validateSpace, 13, 23, 28
- numDeriv2Logf (numDerivLogf), 29
- numDerivLogf, 3, 7, 11, 29
- order, 38
- param, 6, 10, 11, 31, 39
- plot.design, 8, 10, 34
- print.nint\_space, 35
- reduce, 8, 36
- rowmatch, 36
- roworder, 37
- seq, 38
- seq1, 38
- Simplify, 3
- update.param, 8, 31, 39
- Wynn, 8, 10, 40