Package 'calculus'

March 23, 2020

Type Package

Title High Dimensional Numerical and Symbolic Calculus

Version 0.2.1

Description Efficient C++ optimized functions for numerical and symbolic calculus. It includes basic symbolic arithmetic, tensor calculus, Einstein summing convention, fast computation of the Levi-Civita symbol and generalized Kronecker delta, Taylor series expansion, multivariate Hermite polynomials, accurate high-order derivatives, differential operators (Gradient, Jacobian, Hessian, Divergence, Curl, Laplacian) and numerical integration in arbitrary orthogonal coordinate systems: cartesian, polar, spherical, cylindrical, parabolic or user defined by custom scale factors.

License GPL-3

URL https://github.com/emanuele-guidotti/calculus

BugReports https://github.com/emanuele-guidotti/calculus/issues

LinkingTo Rcpp

Imports Rcpp (>= 1.0.1)

Suggests cubature

RoxygenNote 7.0.2

NeedsCompilation yes

Author Emanuele Guidotti [aut, cre] (<https://orcid.org/0000-0002-8961-6623>)

Maintainer Emanuele Guidotti <emanuele.guidotti@unine.ch>

Repository CRAN

Date/Publication 2020-03-23 11:40:02 UTC

R topics documented:

`c2e` *Character to Expression*

Description

Converts characters to expressions

Usage

```
c2e(x)
```

Arguments

`x`	characters

Value

expressions.

See Also

e2c

Examples

```
# convert characters
c2e('a')

# convert array of characters
c2e(array('a', dim = c(2,2)))
```

---

cross *Numerical and Symbolic Cross Product*

Description

Computes the generic cross product of N-1 vectors of length N.

Usage

```
cross(...)

x %cross% y
```

Arguments

...	N-1 vectors of length N.
x	3-d vector
y	3-d vector

Value

N-dimensional vector orthogonal to the N-1 vectors.

Functions

- cross: N-d cross product
- %cross%: 3-d cross product

Examples

```
# canonical basis 3-d
c(1,0,0) %cross% c(0,1,0)

# canonical basis 4-d
cross(c(1,0,0,0), c(0,1,0,0), c(0,0,0,1))
```

curl *Numerical and Symbolic Curl*

Description

Computes the curl of functions, expressions and characters.

Usage

```
curl(f, var, accuracy = 2, stepsize = NULL, coordinates = "cartesian")

f %curl% var
```

Arguments

f	function, expression or character array.
var	character vector, giving the variable names with respect to which derivatives will be computed. If a named vector is provided, derivatives will be computed at that point.
accuracy	accuracy degree for numerical derivatives.
stepsize	finite differences stepsize for numerical derivatives. Auto-optimized by default.
coordinates	coordinate system to use. One of: `cartesian`, `polar`, `spherical`, `cylindrical`, `parabolic`, `parabolic-cylindrical` or a character vector of scale factors for each varibale.

Value

curl array.

Functions

- `curl`: arbitrary coordinate system
- `%curl%`: cartesian coordinates

Examples

```
# curl of a vector field
f <- c('x*y','y*z','x*z')
curl(f, var = c('x','y','z'))
f %curl% c('x','y','z')

# irrotational vector field
f <- c('x','-y','z')
curl(f, var = c('x','y','z'))
f %curl% c('x','y','z')

# numerical curl of a vector field
```

```
f <- c(function(x,y,z) x*y, function(x,y,z) y*z, function(x,y,z) x*z)
curl(f, var = c('x'=1,'y'=1,'z'=1))
f %curl% c('x'=1,'y'=1,'z'=1)

# curl of array of vector fields
f1 <- c('x*y','y*z','z*x')
f2 <- c('x','-y','z')
a <- matrix(c(f1,f2), nrow = 2, byrow = TRUE)
curl(a, var = c('x','y','z'))
a %curl% c('x','y','z')

# curl in polar coordinates
f <- c('sqrt(r)/10','sqrt(r)')
curl(f, var = c('r','phi'), coordinates = 'polar')
```

`derivative` *Numerical and Symbolic Derivatives*

Description

Computes symbolic derivatives based on the `D` function, or accurate and reliable numerical derivatives based on finite differences.

Usage

```
derivative(
  f,
  var = "x",
  order = 1,
  accuracy = 2,
  stepsize = NULL,
  deparse = TRUE
)
```

Arguments

`f`	function, expression or character array.
`var`	character vector, giving the variable names with respect to which derivatives will be computed. If a named vector is provided, derivatives will be computed at that point. See examples.
`order`	integer vector, giving the differentiation order for each variable. See details.
`accuracy`	accuracy degree for numerical derivatives.
`stepsize`	finite differences stepsize for numerical derivatives. Auto-optimized by default.
`deparse`	logical. Return character instead of expression or call?

Details

The function behaves differently depending on the length of the `order` argument.

If `order` is of length 1, then the n-th order derivative is computed for each function with respect to each variable.

$$D = \partial^{(n)} \otimes F \to D_{i,..,j,k,..,l} = \partial^{(n)}_{k,..,l} F_{i,..,j}$$

where F is the tensor of functions and ∂ is the tensor of variable names with respect to which the n-th order derivatives will be computed.

If `order` matches the length of `var`, then it is assumed that the differentiation order is provided for each variable. In this case, each function will be derived n_i times with respect to the i-th variable, for each of the j variables.

$$D = \partial_1^{(n_1)} \partial_{...}^{(...)} \partial_i^{(n_i)} \partial_{...}^{(...)} \partial_j^{(n_j)} F$$

where F is the tensor of functions to differentiate.

If `var` is a named vector, e.g. `c(x = 0,y = 0)`, derivatives will be computed at that point. Note that if `f` is a function, then `var` must be a named vector giving the point at which the numerical derivatives will be computed.

Value

array of derivatives.

Examples

```
# derive f with respect to x
derivative(f = "sin(x)", var = "x")

# derive f with respect to x and evaluate in x = 0
derivative(f = "sin(x)", var = c("x" = 0))

# derive f twice with respect to x
derivative(f = "sin(x)", var = "x", order = 2)

# derive f once with respect to x, and twice with respect to y
derivative(f = "y^2*sin(x)", var = c("x","y"), order = c(1,2))

# compute the gradient of f with respect to (x,y)
derivative(f = "y*sin(x)", var = c("x","y"))

# compute the Jacobian of f with respect to (x,y)
f <- c("y*sin(x)", "x*cos(y)")
derivative(f = f, var = c("x","y"))

# compute the Hessian of f with respect to (x,y)
g <- derivative(f = "y^2*sin(x)", var = c("x","y"))
derivative(f = g, var = c("x","y"))

# compute the Jacobian of f with respect to (x,y) and evaluate in (0,0)
f1 <- function(x, y) y*sin(x)
f2 <- function(x, y) x*cos(y)
derivative(f = c(f1, f2), var = c("x"=0,"y"=0))
```

det *Numerical and Symbolic Determinant*

Description

Calculates the determinant of a matrix.

Usage

```
det(x)
```

Arguments

x	numeric or character matrix.

Value

numeric or character determinant.

Examples

```
# numeric matrix
x <- matrix(1:4, nrow = 2)
det(x)

# symbolic matrix
x <- matrix(letters[1:4], nrow = 2)
det(x)
```

diag *Tensor Diagonals*

Description

Extracts or replace the diagonal of a tensor, or construct a diagonal tensor.

Usage

```
diag(x, dim = 2, value = 1)

diag(x) <- value
```

Arguments

x	array, vector or integer.
dim	the dimension of the tensor.
value	the value for the diagonal elements.

Value

array diagonals.

Functions

- diag: get diagonals.
- diag<-: set diagonals.

Examples

```
# construct a diagonal 2x2 matrix
diag(2)

# construct a diagonal 2x2x2 tensor
diag(2, dim = 3)

# construct a diagonal 2x2x2 tensor with values 3 and 4
diag(2, dim = 3, value = c(3,4))

# construct a diagonal 3x3 matrix with values 1,2,3
diag(1:3)

# extract diagonals
x <- diag(1:4, dim = 3)
diag(x)

# replace diagonals
x <- diag(1:4, dim = 3)
diag(x) <- c(5,6,7,8)
x
```

---

divergence *Numerical and Symbolic Divergence*

Description

Computes the divergence of functions, expressions and characters.

Usage

```
divergence(f, var, accuracy = 2, stepsize = NULL, coordinates = "cartesian")

f %divergence% var
```

Arguments

f	function, expression or character array.
var	character vector, giving the variable names with respect to which derivatives will be computed. If a named vector is provided, derivatives will be computed at that point.
accuracy	accuracy degree for numerical derivatives.
stepsize	finite differences stepsize for numerical derivatives. Auto-optimized by default.
coordinates	coordinate system to use. One of: `cartesian`, `polar`, `spherical`, `cylindrical`, `parabolic`, `parabolic-cylindrical` or a character vector of scale factors for each varibale.

Value

divergence array.

Functions

- `divergence`: arbitrary coordinate system
- `%divergence%`: cartesian coordinates

Examples

```
# divergence of a vector field
f <- c('x^2','y^3','z^4')
divergence(f, var = c('x','y','z'))
f %divergence% c('x','y','z')

# numerical divergence of a vector field
f <- c(function(x,y,z) x^2, function(x,y,z) y^3, function(x,y,z) z^4)
divergence(f, var = c('x'=1,'y'=1,'z'=1))
f %divergence% c('x'=1,'y'=1,'z'=1)

# divergence of array of vector fields
f1 <- c('x^2','y^3','z^4')
f2 <- c('x','y','z')
a <- matrix(c(f1,f2), nrow = 2, byrow = TRUE)
divergence(a, var = c('x','y','z'))
a %divergence% c('x','y','z')

# divergence in polar coordinates
f <- c('sqrt(r)/10','sqrt(r)')
divergence(f, var = c('r','phi'), coordinates = 'polar')
```

e2c *Expression to Character*

Description

Converts expressions to characters

Usage

```
e2c(x)
```

Arguments

x	expressions

Value

characters.

See Also

c2e

Examples

```
# convert expressions
expr <- parse(text = 'a')
e2c(expr)

# convert array of expressions
expr <- array(parse(text = 'a'), dim = c(2,2))
e2c(expr)
```

einstein *Numerical and Symbolic Einstein Summation*

Description

Implements the Einstein notation for summation over repeated indices.

Usage

```
einstein(..., drop = TRUE)
```

Arguments

...	arbitrary number of indexed arrays.
drop	logical. Drop summation indices? If FALSE, keep dummy dimensions.

Value

array.

See Also

index, trace

Examples

```
###################################
# A{i,j} B{j,k,k} C{k,l} D{j,k}
#

a <- array(1:10, dim = c(2,5))
b <- array(1:45, dim = c(5,3,3))
c <- array(1:12, dim = c(3,4))
d <- array(1:15, dim = c(5,3))

index(a) <- c('i','j')
index(b) <- c('j','k','k')
index(c) <- c('k', 'l')
index(d) <- c('j', 'k')

einstein(a,b,c,d)


###################################
# A{i,j} B{j,k}
#

a <- array(letters[1:6], dim = c(2,3))
b <- array(letters[1:12], dim = c(3,4))

index(a) <- c('i','j')
index(b) <- c('j','k')

einstein(a,b)
```

---

evaluate *Numerical Evaluation*

Description

Evaluate an array of characters, expressions or functions.

Usage

```
evaluate(
  x,
  envir = parent.frame(),
 enclos = if (is.list(envir) || is.pairlist(envir)) parent.frame() else baseenv(),
  simplify = TRUE
)
```

Arguments

x	an object to be evaluated: array of characters, expressions or functions.
envir	the environment in which x is to be evaluated. May also be NULL, a list, a data frame, a pairlist or an integer as specified to sys.call.
enclos	relevant when envir is a (pair)list or a data frame. Specifies the enclosure, i.e., where R looks for objects not found in envir. This can be NULL (interpreted as the base package environment, baseenv()) or an environment.
simplify	logical. Simplify the output? If FALSE, return a list.

Value

evaluated object.

Examples

```
##################################
# Evaluate an array of characters
#

x <- array(letters[1:4], dim = c(2,2))

e <- list(a = 1, b = 2, c = 3, d = 4)
evaluate(x, env = e)
evaluate(x, env = e, simplify = FALSE)

e <- list(a = 1:3, b = 2, c = 3, d = 4)
evaluate(x, env = e)
evaluate(x, env = e, simplify = FALSE)


##################################
# Evaluate an array of functions
#

f1 <- function(x,y) sin(x)
f2 <- function(x,y) sin(y)
f3 <- function(x,y) x*y
x <- array(c(f1,f3,f3,f2), dim = c(2,2))

e <- list(x = 0, y = pi/2)
evaluate(x, env = e)
```

```
evaluate(x, env = e, simplify = FALSE)

e <- list(x = c(0, pi/2), y = c(0, pi/2))
evaluate(x, env = e)
evaluate(x, env = e, simplify = FALSE)
```

`gradient` *Numerical and Symbolic Gradient*

Description

Computes the gradient or jacobian of functions, expressions and characters.

Usage

```
gradient(f, var, accuracy = 2, stepsize = NULL, coordinates = "cartesian")

f %gradient% var
```

Arguments

`f`	function, expression or character array.
`var`	character vector, giving the variable names with respect to which derivatives will be computed. If a named vector is provided, derivatives will be computed at that point.
`accuracy`	accuracy degree for numerical derivatives.
`stepsize`	finite differences stepsize for numerical derivatives. Auto-optimized by default.
`coordinates`	coordinate system to use. One of: `cartesian`, `polar`, `spherical`, `cylindrical`, `parabolic`, `parabolic-cylindrical` or a character vector of scale factors for each varibale.

Value

gradient or jacobian array.

Functions

- `gradient`: arbitrary coordinate system
- `%gradient%`: cartesian coordinates

Examples

```
# gradient with respect to x
gradient(f = "sin(x)", var = "x")
"sin(x)" %gradient% "x"

# gradient with respect to x and evaluate in x = 0
gradient(f = "sin(x)", var = c("x" = 0))
"sin(x)" %gradient% c(x=0)

# gradient with respect to (x,y)
gradient(f = "y*sin(x)", var = c("x","y"))
"y*sin(x)" %gradient% c("x","y")

# jacobian with respect to (x,y)
f <- c("y*sin(x)", "x*cos(y)")
gradient(f = f, var = c("x","y"))
f %gradient% c("x","y")

# jacobian with respect to (x,y) and evaluate in (x = 0, y = 0)
f <- c(function(x, y) y*sin(x), function(x, y) x*cos(y))
gradient(f = f, var = c(x=0,y=0))
f %gradient% c(x=0,y=0)

# gradient in spherical coordinates
gradient('r*theta*phi', var = c('r','theta','phi'), coordinates = 'spherical')

# numerical gradient in spherical coordinates
f <- function(r, theta, phi) r*theta*phi
gradient(f, var = c('r'=1, 'theta'=pi/4, 'phi'=pi/4), coordinates = 'spherical')
```

---

hermite *Hermite Polynomials*

Description

Computes univariate and multivariate Hermite polynomials.

Usage

```
hermite(order, sigma = 1, var = "x")
```

Arguments

order	integer. The order of the Hermite polynomial.
sigma	the covariance matrix of the Gaussian kernel.
var	character. The variables of the polynomial.

Details

Hermite polynomials are obtained by successive differentiation of the Gaussian kernel

$$H_\nu(x, \Sigma) = exp\Big(\frac{1}{2}x^\dagger \Sigma x\Big)(-\partial_x)^\nu exp\Big(-\frac{1}{2}x^\dagger \Sigma x\Big)$$

where Σ is a d-dimensional square matrix and $\nu = (\nu_1, ..., \nu_d)$ is the vector representing the order of differentiation for each variable.

Value

list of Hermite polynomials with components

f the Hermite polynomial.

order the order of the Hermite polynomial.

terms data.frame containing the variables, coefficients and degrees of each term in the Hermite polynomial.

Examples

```
# univariate Hermite polynomials up to order 3
hermite(3)

# univariate Hermite polynomials with variable z
hermite(3, var = 'z')

# multivariate Hermite polynomials up to order 2
hermite(order = 2,
        sigma = matrix(c(1,0,0,1), nrow = 2),
        var = c('z1', 'z2'))
```

---

hessian *Numerical and Symbolic Hessian*

Description

Computes the hessian matrix of functions, expressions and characters.

Usage

```
hessian(f, var, accuracy = 2, stepsize = NULL, coordinates = "cartesian")

f %hessian% var
```

Arguments

f	function, expression or character.
var	character vector, giving the variable names with respect to which derivatives will be computed. If a named vector is provided, derivatives will be computed at that point.
accuracy	accuracy degree for numerical derivatives.
stepsize	finite differences stepsize for numerical derivatives. Auto-optimized by default.
coordinates	coordinate system to use. One of: `cartesian`, `polar`, `spherical`, `cylindrical`, `parabolic`, `parabolic-cylindrical` or a character vector of scale factors for each varibale.

Value

hessian matrix.

Functions

- `hessian`: arbitrary coordinate system
- `%hessian%`: cartesian coordinates

Examples

```
# hessian with respect to x
hessian(f = "sin(x)", var = "x")
"sin(x)" %hessian% "x"

# hessian with respect to x and evaluate in x = 0
hessian(f = "sin(x)", var = c("x" = 0))
"sin(x)" %hessian% c(x=0)

# hessian with respect to (x,y)
hessian(f = "y*sin(x)", var = c("x","y"))
"y*sin(x)" %hessian% c("x","y")

# hessian in spherical coordinates
hessian('r*theta*phi', var = c('r','theta','phi'), coordinates = 'spherical')

# numerical hessian in spherical coordinates
f <- function(r, theta, phi) r*theta*phi
hessian(f, var = c('r'=1, 'theta'=pi/4, 'phi'=pi/4), coordinates = 'spherical')
```

index *Einstein Notation Indices*

Description

Get and set indices: names of the array's dimensions. See also einstein.

Usage

```
index(x)

index(x) <- value
```

Arguments

x	array.
value	vector of indices.

Value

array indices.

Functions

- index: get indices.
- index<-: set indices.

See Also

einstein, dim

Examples

```
# define array
a <- array(1, dim = c(1,3,2))

# get indices
index(a)

# set indices
index(a) <- c('i', 'j', 'k')

# get indices
index(a)

# dimensions
dim(a)
```

`integral` *Numerical Integration*

Description

Integrates multidimensional functions, expressions, and characters in arbitrary orthogonal coordinate systems.

Usage

```
integral(
  f,
  bounds,
  relTol = 0.01,
  absTol = 0.001,
  coordinates = "cartesian",
  method = "mc",
  verbose = TRUE,
  ...
)
```

Arguments

`f`	function, expression or character.
`bounds`	list of integration bounds.
`relTol`	relative accuracy requested.
`absTol`	absolute accuracy requested.
`coordinates`	coordinate system to use. One of: `cartesian`, `polar`, `spherical`, `cylindrical`, `parabolic`, `parabolic-cylindrical` or a character vector of scale factors for each varibale.
`method`	the method to use. Should be one of `"mc"`, `"hcubature"`, `"pcubature"`, `"cuhre"`, `"divonne"`, `"suave"` or `"vegas"`. Naive Monte Carlo integration by default. The additional methods require the cubature package to be installed (efficient integration in C).
`verbose`	logical. Print on progress?
`...`	additional arguments passed to the `cubintegrate` function, when method `"hcubature"`, `"pcubature"`, `"cuhre"`, `"divonne"`, `"suave"` or `"vegas"` is used.

Value

list with components

value the final estimate of the integral.

abs.error estimate of the modulus of the absolute error.

cuba cubature output when method `"hcubature"`, `"pcubature"`, `"cuhre"`, `"divonne"`, `"suave"` or `"vegas"` is used.

Examples

```
# integrate character
integral('sin(x)', bounds = list(x = c(0,2*pi)))

# integrate expression
integral(parse(text = 'x'), bounds = list(x = c(0,1)))

# integrate function
integral(function(x) exp(x), bounds = list(x = c(0,1)))

# multivariate integral
integral(function(x,y) x*y, bounds = list(x = c(0,1), y = c(0,1)))

# surface of a sphere
integral('1',
         bounds = list(r = 1, theta = c(0,pi), phi = c(0,2*pi)),
         coordinates = 'spherical')

# volume of a sphere
integral('1',
         bounds = list(r = c(0,1), theta = c(0,pi), phi = c(0,2*pi)),
         coordinates = 'spherical')

## Not run:
# efficient integration in C (requires the cubature package to be installed)
integral('1',
        bounds = list(r = c(0,1), theta = c(0,pi), phi = c(0,2*pi)),
        coordinates = 'spherical',
        method = 'cuhre',
        relTol = 1e-06,
        absTol = 1e-12)

## End(Not run)

##################################
# Electric charge contained in a region of space
# (see divergence theorem and Maxwell's equations)
#

# electric potential of unitary point charge
V <- '1/(4*pi*r)'

# electric field
E <- -1 %prod% gradient(V, c('r', 'theta', 'phi'), coordinates = 'spherical')

# electric charge
integral(E[1],
         bounds = list(r = 1, theta = c(0,pi), phi = c(0,2*pi)),
         coordinates = 'spherical')
```

inverse *Numeric and Symbolic Matrix Inverse*

Description

Calculates the inverse of a matrix.

Usage

```
inverse(x)
```

Arguments

x	numeric or character matrix.

Value

numeric or character matrix.

Examples

```
# numeric matrix
x <- matrix(1:4, nrow = 2)
inverse(x)

# symbolic matrix
x <- matrix(letters[1:4], nrow = 2)
inverse(x)
```

kronecker *Generalized Kronecker Delta*

Description

Computes the Generalized Kronecker Delta.

Usage

```
kronecker(n, p = 1)
```

Arguments

n	number of elements for each dimension.
p	order of the generalized Kronecker delta, p=1 for the standard Kronecker delta.

Value

array representing the generalized Kronecker delta tensor.

Examples

```
# Kronecker delta 3x3
kronecker(3)

# generalized Kronecker delta 3x3 of order 2 -> 3x3 x 3x3
kronecker(3, p = 2)
```

`laplacian` *Numerical and Symbolic Laplacian*

Description

Computes the laplacian of functions, expressions and characters.

Usage

```
laplacian(f, var, accuracy = 2, stepsize = NULL, coordinates = "cartesian")

f %laplacian% var
```

Arguments

`f`	function, expression or character array.
`var`	character vector, giving the variable names with respect to which derivatives will be computed. If a named vector is provided, derivatives will be computed at that point.
`accuracy`	accuracy degree for numerical derivatives.
`stepsize`	finite differences stepsize for numerical derivatives. Auto-optimized by default.
`coordinates`	coordinate system to use. One of: `cartesian`, `polar`, `spherical`, `cylindrical`, `parabolic`, `parabolic-cylindrical` or a character vector of scale factors for each varibale.

Value

laplacian array.

Functions

- `laplacian`: arbitrary coordinate system
- `%laplacian%`: cartesian coordinates

Examples

```
# laplacian of a scalar field
f <- 'x^2+y^2+z^2'
laplacian(f, var = c('x','y','z'))
f %laplacian% c('x','y','z')

# laplacian of scalar fields
f <- c('x^2','y^3','z^4')
laplacian(f, var = c('x','y','z'))
f %laplacian% c('x','y','z')

# numerical laplacian of scalar fields
f <- c(function(x,y,z) x^2, function(x,y,z) y^3, function(x,y,z) z^4)
laplacian(f, var = c('x'=1,'y'=1,'z'=1))
f %laplacian% c('x'=1,'y'=1,'z'=1)

# laplacian of array of scalar fields
f1 <- c('x^2','y^3','z^4')
f2 <- c('x','y','z')
a <- matrix(c(f1,f2), nrow = 2, byrow = TRUE)
laplacian(a, var = c('x','y','z'))
a %laplacian% c('x','y','z')

# laplacian in polar coordinates
f <- c('sqrt(r)/10','sqrt(r)')
laplacian(f, var = c('r','phi'), coordinates = 'polar')
```

---

levicivita *Levi-Civita Symbol*

Description

Computes the Levi-Civita totally antisymmetric tensor.

Usage

```
levicivita(n)
```

Arguments

n	dimension

Value

array representing the Levi-Civita tensor.

Examples

```
# Levi-Civita tensor in 2-d
levicivita(2)

# Levi-Civita tensor in 3-d
levicivita(3)
```

`partitions` *Partitions of an Integer*

Description

Provides fast algorithms for generating integer partitions.

Usage

```
partitions(n, max = 0, length = 0, perm = FALSE, fill = FALSE, equal = T)
```

Arguments

`n`	positive integer.
`max`	maximum integer in the partitions.
`length`	maximum number of elements in the partitions.
`perm`	logical. Permute partitions?
`fill`	logical. Fill partitions with zeros to match `length`?
`equal`	logical. Return only partition of `n`? If `FALSE`, partitions of all integers less or equal to `n` are returned.

Value

list of partitions, or data.frame if `length>0` and `fill=TRUE`.

Examples

```
# partitions of 4
partitions(4)

# partitions of 4 and permute
partitions(4, perm = TRUE)

# partitions of 4 with max element 2
partitions(4, max = 2)

# partitions of 4 with 2 elements
partitions(4, length = 2)
```

```
# partitions of 4 with 3 elements, fill with zeros
partitions(4, length = 3, fill = TRUE)

# partitions of 4 with 3 elements, fill with zeros and permute
partitions(4, length = 3, fill = TRUE, perm = TRUE)

# partitions of all integers less or equal to 3
partitions(3, equal = FALSE)

# partitions of all integers less or equal to 3, fill to 2 elements and permute
partitions(3, equal = FALSE, length = 2, fill = TRUE, perm = TRUE)
```

`taylor` *Taylor Series*

Description

Computes the Taylor series for functions, expressions or characters.

Usage

```
taylor(f, var = "x", order = 1, accuracy = 2, stepsize = NULL)
```

Arguments

`f`	function, expression or character
`var`	character. The variables of `f`.
`order`	the order of the Taylor approximation.
`accuracy`	accuracy degree for numerical derivatives.
`stepsize`	finite differences stepsize for numerical derivatives. Auto-optimized by default.

Value

list with components

f the Taylor series.

order the approximation order.

terms data.frame containing the variables, coefficients and degrees of each term in the Taylor series.

Examples

```
# univariate taylor series
taylor('exp(x)', var = 'x', order = 3)

# univariate taylor series of arbitrary functions
taylor(function(x) exp(x), var = 'x', order = 3)

# multivariate taylor series
taylor('sin(x*y)', var = c('x','y'), order = 6)

# multivariate taylor series of arbitrary functions
taylor(function(x,y) sin(x*y), var = c('x','y'), order = 6)
```

---

trace *Tensor Contraction*

Description

Sums over repeated indices in a tensor. Can be seen as a generalization of the trace.

Usage

```
trace(x, i = NULL, drop = TRUE)
```

Arguments

x	array.
i	subset of repeated indices to sum up. If NULL, the tensor contraction takes place on all repeated indices of x.
drop	logical. Drop summation indices? If FALSE, keep dummy dimensions.

Value

array.

See Also

index, einstein

Examples

```
# trace of numeric matrix
x <- matrix(1:4, nrow = 2)
trace(x)

# trace of character matrix
x <- matrix(letters[1:4], nrow = 2)
```

```
trace(x)

# trace of a tensor (sum over diagonals)
x <- array(1:27, dim = c(3,3,3))
trace(x)

# tensor contraction over repeated indices
x <- array(1:27, dim = c(3,3,3))
index(x) <- c('i','i','j')
trace(x)

# tensor contraction over specific indices only
x <- array(1:16, dim = c(2,2,2,2))
index(x) <- c('i','i','k','k')
trace(x, i = 'k')

# tensor contraction keeping dummy dimensions
x <- array(letters[1:16], dim = c(2,2,2,2))
index(x) <- c('i','i','k','k')
trace(x, drop = FALSE)
```

---

wrap *Wrap Character*

Description

Wraps characters in round brackets.

Usage

```
wrap(x)
```

Arguments

x	characters

Details

Characters are automatically wrapped when performing basic symbolic operations to prevent unwanted results. E.g.:

$$a + b * c + d$$

instead of

$$(a + b) * (c + d)$$

To disable this behaviour run `options(calculus.auto.wrap = FALSE)`.

Value

wrapped characters.

Examples

```
# wrap characters
wrap('a+b')

# wrap array of characters
wrap(array(letters[1:9], dim = c(3,3)))
```

`%diff%` *Numerical and Symbolic Difference*

Description

Substracts numeric or character arrays element by element.

Usage

```
x %diff% y
```

Arguments

x	character or numeric array.
y	character or numeric array.

Value

character or numeric array.

Examples

```
# diff vector
x <- c("a+1","b+2")
x %diff% x

# diff matrix
x <- matrix(letters[1:4], ncol = 2)
x %diff% x

# diff array
x <- array(letters[1:12], dim = c(2,2,3))
y <- array(1:12, dim = c(2,2,3))
x %diff% x
y %diff% y
x %diff% y
```

%div% *Numerical and Symbolic Division*

Description

Divide numeric or character arrays element by element.

Usage

```
x %div% y
```

Arguments

x	character or numeric array.
y	character or numeric array.

Value

character or numeric array.

Examples

```
# div vector
x <- c("a+1","b+2")
x %div% x

# div matrix
x <- matrix(letters[1:4], ncol = 2)
x %div% x

# div array
x <- array(letters[1:12], dim = c(2,2,3))
y <- array(1:12, dim = c(2,2,3))
x %div% x
y %div% y
x %div% y
```

%dot% *Numerical and Symbolic Dot Product*

Description

Computes the inner product on the last dimensions of two character or numeric arrays.

Usage

```
x %dot% y
```

Arguments

x	character or numeric array.
y	character or numeric array.

Value

character or numeric array.

Examples

```
# inner product
x <- array(1:12, dim = c(3,4))
x %dot% x

# inner product on last dimensions
x <- array(1:24, dim = c(3,2,4))
y <- array(letters[1:8], dim = c(2,4))
x %dot% y
y %dot% x
```

%inner% *Numerical and Symbolic Inner Product*

Description

Computes the inner product of two character or numeric arrays.

Usage

```
x %inner% y
```

Arguments

x	character or numeric array.
y	character or numeric array.

Value

character or numeric.

Examples

```
# numeric inner product
x <- array(1:12, dim = c(3,4))
x %inner% x

# symbolic inner product
x <- array(letters[1:12], dim = c(3,4))
x %inner% x
```

`%kronecker%` *Numerical and Symbolic Kronecker Product*

Description

Computes the Kronecker product of two character or numeric arrays.

Usage

```
x %kronecker% y
```

Arguments

x	character or numeric array.
y	character or numeric array.

Value

character or numeric array.

Examples

```
# numeric Kronecker product
c(1,2) %kronecker% c(2,3)

# symbolic Kronecker product
array(1:4, dim = c(2,2)) %kronecker% c('c','d')
```

`%matrix%` *Numerical and Symbolic Matrix Product*

Description

Multiplies two character or numeric matrices, if they are conformable. If one argument is a vector, it will be promoted to either a row or column matrix to make the two arguments conformable. If both are vectors of the same length, it will return the inner product (as a matrix).

Usage

```
x %matrix% y
```

Arguments

x	character or numeric matrix.
y	character or numeric matrix.

Value

character or numeric matrix.

Examples

```
# numeric inner product
x <- 1:4
x %matrix% x

# symbolic inner product
x <- letters[1:4]
x %matrix% x

# matrix products
x <- letters[1:4]
y <- diag(4)
z <- array(1:12, dim = c(4,3))
y %matrix% z
y %matrix% x
x %matrix% z
```

%outer% *Numerical and Symbolic Outer Product*

Description

Computes the outer product of two character or numeric arrays.

Usage

```
x %outer% y
```

Arguments

x	character or numeric array.
y	character or numeric array.

Value

character or numeric array.

Examples

```
# numeric outer product
c(1,2) %outer% c(2,3)

# symbolic outer product
c('a','b') %outer% c('c','d') %outer% c('e','f')
```

%prod% *Numerical and Symbolic Product*

Description

Multiplies numeric or character arrays element by element.

Usage

```
x %prod% y
```

Arguments

x	character or numeric array.
y	character or numeric array.

Value

character or numeric array.

Examples

```
# prod vector
x <- c("a+1","b+2")
x %prod% x

# prod matrix
x <- matrix(letters[1:4], ncol = 2)
x %prod% x

# prod array
x <- array(letters[1:12], dim = c(2,2,3))
y <- array(1:12, dim = c(2,2,3))
x %prod% x
y %prod% y
x %prod% y
```

%sum% *Numerical and Symbolic Sum*

Description

Adds numeric or character arrays element by element.

Usage

```
x %sum% y
```

Arguments

x	character or numeric array.
y	character or numeric array.

Value

character or numeric array.

Examples

```
# sum vector
x <- c("a+1","b+2")
x %sum% x

# sum matrix
x <- matrix(letters[1:4], ncol = 2)
```

```
x %sum% x

# sum array
x <- array(letters[1:12], dim = c(2,2,3))
y <- array(1:12, dim = c(2,2,3))
x %sum% x
y %sum% y
x %sum% y
```

Index