Package 'calculus'

August 21, 2022

Type Package

Title High Dimensional Numerical and Symbolic Calculus

Version 0.3.4

Description Efficient C++ optimized functions for numerical and symbolic calculus as described in Guidotti (2020) <arXiv:2101.00086>. It includes basic arithmetic, tensor calculus, Einstein summing convention, fast computation of the Levi-Civita symbol and generalized Kronecker delta, Taylor series expansion, multivariate Hermite polynomials, high-order derivatives, ordinary differential equations, 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://calculus.guidotti.dev

BugReports https://github.com/eguidotti/calculus/issues

LinkingTo Rcpp

Imports Rcpp (>= 1.0.1)

Suggests cubature, testthat, knitr, rmarkdown

RoxygenNote 7.2.1

VignetteBuilder knitr

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 2022-08-20 22:10:17 UTC

R topics documented:

`c2e` *Characters to Expressions*

Description

Converts `characters` to `expressions`.

Usage

```
c2e(x)
```

Arguments

`x`	`character`.

Value

expression.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other utilities: e2c(), evaluate(), wrap()

Examples

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

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

---

contraction *Numerical and Symbolic Tensor Contraction*

Description

Sums over repeated indices in an array.

Usage

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

Arguments

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

Value

array.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other tensor algebra: delta(), diagonal(), einstein(), epsilon(), index()

Examples

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

### tensor trace
x <- array(1:27, dim = c(3,3,3))
contraction(x)

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

#### tensor contraction over specific repeated indices only
x <- array(1:16, dim = c(2,2,2,2))
index(x) <- c("i","i","k","k")
contraction(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")
contraction(x, drop = FALSE)
```

---

cross *Numerical and Symbolic Cross Product*

Description

Computes the cross product of $n-1$ vectors of length n.

Usage

```
cross(...)

x %cross% y
```

Arguments

...	$n-1$ vectors of length n.
x	numeric or character vector of length 3.
y	numeric or character vector of length 3.

Value

n-dimensional vector orthogonal to the $n-1$ vectors.

Functions

- `x %cross% y`: binary operator for 3-dimensional cross products.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

Examples

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

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

### symbolic cross product 3-d
c(1,0,0) %cross% c(0,1,0)

### symbolic cross product 3-d
c("a","b","c") %cross% c(0,0,1)
```

`curl` *Numerical and Symbolic Curl*

Description

Computes the numerical curl of `functions` or the symbolic curl of `characters` in arbitrary orthogonal coordinate systems.

Usage

```
curl(
  f,
  var,
  params = list(),
  coordinates = "cartesian",
  accuracy = 4,
  stepsize = NULL,
  drop = TRUE
)

f %curl% var
```

Arguments

f	array of characters or a function returning a numeric array.
var	vector giving the variable names with respect to which the derivatives are to be computed and/or the point where the derivatives are to be evaluated. See derivative.
params	list of additional parameters passed to f.
coordinates	coordinate system to use. One of: cartesian, polar, spherical, cylindrical, parabolic, parabolic-cylindrical or a vector of scale factors for each variable.
accuracy	degree of accuracy for numerical derivatives.
stepsize	finite differences stepsize for numerical derivatives. It is based on the precision of the machine by default.
drop	if TRUE, return the curl as a vector and not as an array for vector-valued functions.

Details

The curl of a vector-valued function F_i at a point is represented by a vector whose length and direction denote the magnitude and axis of the maximum circulation. In 2 dimensions, the curl is computed in arbitrary orthogonal coordinate systems using the scale factors h_i and the Levi-Civita symbol epsilon:

$$\nabla \times F = \frac{1}{h_1 h_2} \sum_{ij} \epsilon_{ij} \partial_i \left(h_j F_j \right) = \frac{1}{h_1 h_2} \left(\partial_1 \left(h_2 F_2 \right) - \partial_2 \left(h_1 F_1 \right) \right)$$

In 3 dimensions:

$$(\nabla \times F)_k = \frac{h_k}{J} \sum_{ij} \epsilon_{ijk} \partial_i \left(h_j F_j \right)$$

where $J = \prod_i h_i$. In $m+2$ dimensions, the curl is implemented in such a way that the formula reduces correctly to the previous cases for $m = 0$ and $m = 1$:

$$(\nabla \times F)_{k_1 \ldots k_m} = \frac{h_{k_1} \cdots h_{k_m}}{J} \sum_{ij} \epsilon_{ijk_1 \ldots k_m} \partial_i \left(h_j F_j \right)$$

When F is an array of vector-valued functions $F_{d_1,\ldots,d_n,j}$ the curl is computed for each vector:

$$(\nabla \times F)_{d_1 \ldots d_n, k_1 \ldots k_m} = \frac{h_{k_1} \cdots h_{k_m}}{J} \sum_{ij} \epsilon_{ijk_1 \ldots k_m} \partial_i \left(h_j F_{d_1 \ldots d_n, j} \right)$$

Value

Vector for vector-valued functions when drop=TRUE, array otherwise.

Functions

- `f %curl% var`: binary operator with default parameters.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other differential operators: `derivative()`, `divergence()`, `gradient()`, `hessian()`, `jacobian()`, `laplacian()`

Examples

```
### symbolic curl of a 2-d vector field
f <- c("x^3*y^2","x")
curl(f, var = c("x","y"))

### numerical curl of a 2-d vector field in (x=1, y=1)
f <- function(x,y) c(x^3*y^2, x)
curl(f, var = c(x=1, y=1))

### numerical curl of a 3-d vector field in (x=1, y=1, z=1)
f <- function(x,y,z) c(x^3*y^2, x, z)
curl(f, var = c(x=1, y=1, z=1))

### vectorized interface
f <- function(x) c(x[1]^3*x[2]^2, x[1], x[3])
curl(f, var = c(1,1,1))

### symbolic array of vector-valued 3-d functions
f <- array(c("x*y","x","y*z","y","x*z","z"), dim = c(2,3))
curl(f, var = c("x","y","z"))

### numeric array of vector-valued 3-d functions in (x=1, y=1, z=1)
f <- function(x,y,z) array(c(x*y,x,y*z,y,x*z,z), dim = c(2,3))
curl(f, var = c(x=1, y=1, z=1))

### binary operator
c("x*y","y*z","x*z") %curl% c("x","y","z")
```

`delta` *Generalized Kronecker Delta*

Description

Computes the Generalized Kronecker Delta.

Usage

```
delta(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.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other tensor algebra: contraction(), diagonal(), einstein(), epsilon(), index()

Examples

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

### generalized Kronecker delta 3x3 of order 2
delta(3, p = 2)
```

---

derivative *Numerical and Symbolic Derivatives*

Description

Computes symbolic derivatives based on the D function, or numerical derivatives based on finite differences.

Usage

```
derivative(
  f,
  var,
  params = list(),
  order = 1,
  accuracy = 4,
  stepsize = NULL,
  drop = TRUE,
```

```
  deparse = TRUE
)
```

Arguments

f	array of characters or a function returning a numeric array.
var	vector giving the variable names with respect to which the derivatives are to be computed and/or the point where the derivatives are to be evaluated. See details.
params	list of additional parameters passed to f.
order	integer vector, giving the differentiation order for each variable. See details.
accuracy	degree of accuracy for numerical derivatives.
stepsize	finite differences stepsize for numerical derivatives. It is based on the precision of the machine by default.
drop	if TRUE, return the array of derivatives without adding a dummy dimension when order is of length 1.
deparse	if TRUE, return character instead of expression.

Details

The function behaves differently depending on the arguents order, the order of differentiation, and var, the variable names with respect to which the derivatives are computed.

When multiple variables are provided and order is a single integer n, then the n-th order derivative is computed for each element of f with respect to each variable:

$$D = \partial^{(n)} \otimes F$$

that is:

$$D_{i,\ldots,j,k} = \partial_k^{(n)} F_{i,\ldots,j}$$

where F is the array of functions and $\partial_k^{(n)}$ denotes the n-th order partial derivative with respect to the k-th variable.

When order matches the length of var, it is assumed that the differentiation order is provided for each variable. In this case, each element is derived n_k times with respect to the k-th variable, for each of the m variables.

$$D_{i,\ldots,j} = \partial_1^{(n_1)} \cdots \partial_m^{(n_m)} F_{i,\ldots,j}$$

The same applies when order is a named vector giving the differentiation order for each variable. For example, order = c(x=1, y=2) differentiates once with respect to x and twice with respect to y. A call with order = c(x=1, y=0) is equivalent to order = c(x=1).

To compute numerical derivatives or to evaluate symbolic derivatives at a point, the function accepts a named vector for the argument var; e.g. var = c(x=1, y=2) evaluates the derivatives in $x = 1$ and $y = 2$. For functions where the first argument is used as a parameter vector, var should be a numeric vector indicating the point at which the derivatives are to be calculated.

Value

array.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other derivatives: taylor()

Other differential operators: curl(), divergence(), gradient(), hessian(), jacobian(), laplacian()

Examples

```
### symbolic derivatives
derivative(f = "sin(x)", var = "x")

### numerical derivatives
f <- function(x) sin(x)
derivative(f = f, var = c(x=0))

### higher order derivatives
f <- function(x) sin(x)
derivative(f = f, var = c(x=0), order = 3)

### multivariate functions
##  - derive once with respect to x
##  - derive twice with respect to y
##  - evaluate in x=0 and y=0
f <- function(x, y) y^2*sin(x)
derivative(f = f, var = c(x=0, y=0), order = c(1,2))

### vector-valued functions
##  - derive each element twice with respect to each variable
##  - evaluate in x=0 and y=0
f <- function(x, y) c(x^2, y^2)
derivative(f, var = c(x=0, y=0), order = 2)

### vectorized interface
f <- function(x) c(sum(x), prod(x))
derivative(f, var = c(0,0,0), order = 1)
```

---

diagonal *Tensor Diagonals*

Description

Functions to extract or replace the diagonals of an array, or construct a diagonal array.

Usage

```
diagonal(x = 1, dim = rep(2, 2))

diagonal(x) <- value
```

Arguments

x	an array from which to extract the diagonals, or a vector giving the diagonal values to construct the array.
dim	the dimensions of the (square) array to construct when x is a vector.
value	vector giving the values of the diagonal entries.

Value

Vector of the diagonal entries of x if x is an array. If x is a vector, returns the diagonal array with the entries given by x.

Functions

- diagonal(x) <- value: set diagonals.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other tensor algebra: contraction(), delta(), einstein(), epsilon(), index()

Examples

```
### 3x3 matrix
diagonal(x = 1, dim = c(3,3))

### 2x2x2 array
diagonal(x = 1:2, dim = c(2,2,2))

### extract diagonals
x <- diagonal(1:5, dim = c(5,5,5))
diagonal(x)

### set diagonals
x <- array(0, dim = c(2,2,2))
diagonal(x) <- 1:2
x
```

`divergence` *Numerical and Symbolic Divergence*

Description

Computes the numerical divergence of `functions` or the symbolic divergence of `characters` in arbitrary orthogonal coordinate systems.

Usage

```
divergence(
  f,
  var,
  params = list(),
  coordinates = "cartesian",
  accuracy = 4,
  stepsize = NULL,
  drop = TRUE
)

f %divergence% var
```

Arguments

`f`	array of `characters` or a `function` returning a `numeric` array.
`var`	vector giving the variable names with respect to which the derivatives are to be computed and/or the point where the derivatives are to be evaluated. See `derivative`.
`params`	`list` of additional parameters passed to `f`.
`coordinates`	coordinate system to use. One of: `cartesian`, `polar`, `spherical`, `cylindrical`, `parabolic`, `parabolic-cylindrical` or a vector of scale factors for each variable.
`accuracy`	degree of accuracy for numerical derivatives.
`stepsize`	finite differences stepsize for numerical derivatives. It is based on the precision of the machine by default.
`drop`	if `TRUE`, return the divergence as a scalar and not as an `array` for vector-valued functions.

Details

The divergence of a vector-valued function F_i produces a scalar value $\nabla \cdot F$ representing the volume density of the outward flux of the vector field from an infinitesimal volume around a given point. The `divergence` is computed in arbitrary orthogonal coordinate systems using the scale factors h_i:

$$\nabla \cdot F = \frac{1}{J} \sum_i \partial_i \left(\frac{J}{h_i} F_i \right)$$

where $J = \prod_i h_i$. When F is an `array` of vector-valued functions $F_{d_1 \ldots d_n, i}$, the `divergence` is computed for each vector:

$$(\nabla \cdot F)_{d_1 \ldots d_n} = \frac{1}{J} \sum_i \partial_i \left(\frac{J}{h_i} F_{d_1 \ldots d_n, i} \right)$$

Value

Scalar for vector-valued functions when `drop=TRUE`, `array` otherwise.

Functions

- `f %divergence% var`: binary operator with default parameters.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other differential operators: `curl()`, `derivative()`, `gradient()`, `hessian()`, `jacobian()`, `laplacian()`

Examples

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

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

### vectorized interface
f <- function(x) c(x[1]^2, x[2]^3, x[3]^4)
divergence(f, var = c(1,1,1))

### symbolic array of vector-valued 3-d functions
f <- array(c("x^2","x","y^2","y","z^2","z"), dim = c(2,3))
divergence(f, var = c("x","y","z"))

### numeric array of vector-valued 3-d functions in (x=0, y=0, z=0)
f <- function(x,y,z) array(c(x^2,x,y^2,y,z^2,z), dim = c(2,3))
divergence(f, var = c(x=0, y=0, z=0))

### binary operator
c("x^2","y^3","z^4") %divergence% c("x","y","z")
```

e2c *Expressions to Characters*

Description

Converts expressions to characters.

Usage

```
e2c(x)
```

Arguments

x	expression.

Value

character.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other utilities: c2e(), evaluate(), wrap()

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`. See `index`.
`drop`	`logical`. Drop summation indices? If `FALSE`, keep dummy dimensions.

Value

`array`.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other tensor algebra: `contraction()`, `delta()`, `diagonal()`, `epsilon()`, `index()`

Examples

```
### A{i,j} B{j,k}
a <- array(letters[1:6], dim = c(i=2, j=3))
b <- array(letters[1:3], dim = c(j=3, k=1))
einstein(a,b)

### A{i,j} B{j,k,k} C{k,l} D{j,k}
a <- array(1:10, dim = c(i=2, j=5))
b <- array(1:45, dim = c(j=5, k=3, k=3))
c <- array(1:12, dim = c(k=3, l=4))
d <- array(1:15, dim = c(j=5, k=3))
einstein(a,b,c,d)
```

epsilon *Levi-Civita Symbol*

Description

Computes the Levi-Civita totally antisymmetric tensor.

Usage

```
epsilon(n)
```

Arguments

n	number of dimensions.

Value

`array` representing the Levi-Civita symbol.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other tensor algebra: `contraction()`, `delta()`, `diagonal()`, `einstein()`, `index()`

Examples

```
### Levi-Civita symbol in 2 dimensions
epsilon(2)

### Levi-Civita symbol in 3 dimensions
epsilon(3)
```

evaluate *Evaluate Characters and Expressions*

Description

Evaluates an array of characters or expressions.

Usage

```
evaluate(f, var, params = list(), vectorize = TRUE)
```

Arguments

f	array of characters or expressions to be evaluated.
var	named vector or data.frame in which f is to be evaluated.
params	list of additional parameters passed to f.
vectorize	logical. Use vectorization? If TRUE, it can significantly boost performance but f needs to handle the vector of inputs appropriately.

Value

Evaluated object. When var is a named vector, the return is an array with the same dimensions of f. When var is a data.frame, the return is a matrix with columns corresponding to the entries of f and rows corresponding to the rows of var.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other utilities: c2e(), e2c(), wrap()

Examples

```
### single evaluation
f <- array(letters[1:4], dim = c(2,2))
var <- c(a = 1, b = 2, c = 3, d = 4)
evaluate(f, var)

### multiple evaluation
f <- array(letters[1:4], dim = c(2,2))
var <- data.frame(a = 1:3, b = 2:4, c = 3:5, d = 4:6)
evaluate(f, var)

### multiple evaluation with additional parameters
f <- "a*sum(x)"
```

```
var <- data.frame(a = 1:3)
params <- list(x = 1:3)
evaluate(f, var, params)

### multiple evaluation of non-vectorized expressions
f <- "a*myf(x)"
myf <- function(x) if(x>0) 1 else -1
var <- data.frame(a = 1:3, x = -1:1)
evaluate(f, var, params = list(myf = myf), vectorize = FALSE)
```

`gradient` *Numerical and Symbolic Gradient*

Description

Computes the numerical gradient of `functions` or the symbolic gradient of `characters` in arbitrary orthogonal coordinate systems.

Usage

```
gradient(
  f,
  var,
  params = list(),
  coordinates = "cartesian",
  accuracy = 4,
  stepsize = NULL,
  drop = TRUE
)

f %gradient% var
```

Arguments

`f`	array of `characters` or a `function` returning a `numeric` array.
`var`	vector giving the variable names with respect to which the derivatives are to be computed and/or the point where the derivatives are to be evaluated. See `derivative`.
`params`	`list` of additional parameters passed to `f`.
`coordinates`	coordinate system to use. One of: `cartesian`, `polar`, `spherical`, `cylindrical`, `parabolic`, `parabolic-cylindrical` or a vector of scale factors for each variable.
`accuracy`	degree of accuracy for numerical derivatives.
`stepsize`	finite differences stepsize for numerical derivatives. It is based on the precision of the machine by default.
`drop`	if `TRUE`, return the gradient as a vector and not as an `array` for scalar-valued functions.

Details

The gradient of a scalar-valued function F is the vector $(\nabla F)_i$ whose components are the partial derivatives of F with respect to each variable i. The `gradient` is computed in arbitrary orthogonal coordinate systems using the scale factors h_i:

$$(\nabla F)_i = \frac{1}{h_i}\partial_i F$$

When the function F is a tensor-valued function $F_{d_1,\ldots,d_n}$, the `gradient` is computed for each scalar component. In particular, it becomes the Jacobian matrix for vector-valued function.

$$(\nabla F_{d_1,\ldots,d_n})_i = \frac{1}{h_i}\partial_i F_{d_1,\ldots,d_n}$$

Value

Gradient vector for scalar-valued functions when `drop=TRUE`, `array` otherwise.

Functions

- `f %gradient% var`: binary operator with default parameters.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other differential operators: `curl()`, `derivative()`, `divergence()`, `hessian()`, `jacobian()`, `laplacian()`

Examples

```
### symbolic gradient
gradient("x*y*z", var = c("x", "y", "z"))

### numerical gradient in (x=1, y=2, z=3)
f <- function(x, y, z) x*y*z
gradient(f = f, var = c(x=1, y=2, z=3))

### vectorized interface
f <- function(x) x[1]*x[2]*x[3]
gradient(f = f, var = c(1, 2, 3))

### symbolic vector-valued functions
f <- c("y*sin(x)", "x*cos(y)")
gradient(f = f, var = c("x","y"))

### numerical vector-valued functions
f <- function(x) c(sum(x), prod(x))
```

```
gradient(f = f, var = c(0,0,0))

### binary operator
"x*y^2" %gradient% c(x=1, y=3)
```

`hermite` *Hermite Polynomials*

Description

Computes univariate and multivariate Hermite polynomials.

Usage

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

Arguments

`order`	the order of the Hermite polynomial.
`sigma`	the covariance `matrix` of the Gaussian kernel.
`var`	`character` vector giving the variables of the polynomial.
`transform`	`character` vector representing a change of variables. See details.

Details

Hermite polynomials are obtained by differentiation of the Gaussian kernel:

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

where Σ is a d-dimensional square matrix and $\nu = (\nu_1 \dots \nu_d)$ is the vector representing the order of differentiation for each variable $x = (x_1 \dots x_d)$. In the case where $\Sigma = 1$ and $x = x_1$ the formula reduces to the standard univariate Hermite polynomials:

$$H_\nu(x) = e^{\frac{x^2}{2}}(-1)^\nu \frac{d^\nu}{dx^\nu} e^{-\frac{x^2}{2}}$$

If `transform` is not `NULL`, the variables `var` x are replaced with `transform` $f(x)$ to compute the polynomials $H_\nu(f(x),\Sigma)$

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.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other polynomials: `taylor()`

Examples

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

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

### multivariate Hermite polynomials with transformation of variables
hermite(order = 2,
        sigma = matrix(c(1,0,0,1), nrow = 2),
        var = c('z1', 'z2'),
        transform = c('z1+z2','z1-z2'))
```

`hessian` *Numerical and Symbolic Hessian*

Description

Computes the numerical Hessian of `functions` or the symbolic Hessian of `characters`.

Usage

```
hessian(f, var, params = list(), accuracy = 4, stepsize = NULL, drop = TRUE)

f %hessian% var
```

Arguments

`f`	array of `characters` or a `function` returning a `numeric` array.
`var`	vector giving the variable names with respect to which the derivatives are to be computed and/or the point where the derivatives are to be evaluated. See `derivative`.
`params`	`list` of additional parameters passed to `f`.
`accuracy`	degree of accuracy for numerical derivatives.

`stepsize`	finite differences stepsize for numerical derivatives. It is based on the precision of the machine by default.
`drop`	if `TRUE`, return the Hessian as a matrix and not as an `array` for scalar-valued functions.

Details

In Cartesian coordinates, the Hessian of a scalar-valued function F is the square matrix of second-order partial derivatives:

$$(H(F))_{ij} = \partial_{ij} F$$

When the function F is a tensor-valued function $F_{d_1,\ldots,d_n}$, the `hessian` is computed for each scalar component.

$$(H(F))_{d_1 \ldots d_n, ij} = \partial_{ij} F_{d_1 \ldots d_n}$$

It might be tempting to apply the definition of the Hessian as the Jacobian of the gradient to write it in arbitrary orthogonal coordinate systems. However, this results in a Hessian matrix that is not symmetric and ignores the distinction between vector and covectors in tensor analysis. The generalization to arbitrary coordinate system is not currently supported.

Value

Hessian matrix for scalar-valued functions when `drop=TRUE`, `array` otherwise.

Functions

- `f %hessian% var`: binary operator with default parameters.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other differential operators: `curl()`, `derivative()`, `divergence()`, `gradient()`, `jacobian()`, `laplacian()`

Examples

```
### symbolic Hessian
hessian("x*y*z", var = c("x", "y", "z"))

### numerical Hessian in (x=1, y=2, z=3)
f <- function(x, y, z) x*y*z
hessian(f = f, var = c(x=1, y=2, z=3))
```

```
### vectorized interface
f <- function(x) x[1]*x[2]*x[3]
hessian(f = f, var = c(1, 2, 3))

### symbolic vector-valued functions
f <- c("y*sin(x)", "x*cos(y)")
hessian(f = f, var = c("x","y"))

### numerical vector-valued functions
f <- function(x) c(sum(x), prod(x))
hessian(f = f, var = c(0,0,0))

### binary operator
"x*y^2" %hessian% c(x=1, y=3)
```

---

index *Tensor Indices*

Description

Functions to get or set the names of the dimensions of an `array`.

Usage

```
index(x)

index(x) <- value
```

Arguments

`x`	`array`.
`value`	vector of indices.

Value

Vector of indices.

Functions

- `index(x) <- value`: set indices.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other tensor algebra: `contraction()`, `delta()`, `diagonal()`, `einstein()`, `epsilon()`

Examples

```
### array with no indices
x <- array(1, dim = c(1, 3, 2))
index(x)

### indices on initialization
x <- array(1, dim = c(i=1, j=3, k=2))
index(x)

### set indices on the fly
x <- array(1, dim = c(1, 3, 2))
index(x) <- c("i", "j", "k")
index(x)
```

`integral` *Numerical Integration*

Description

Computes the integrals of `functions` or `characters` in arbitrary orthogonal coordinate systems.

Usage

```
integral(
  f,
  bounds,
  params = list(),
  coordinates = "cartesian",
  relTol = 0.001,
  absTol = 1e-12,
  method = NULL,
  vectorize = NULL,
  drop = TRUE,
  verbose = FALSE,
  ...
)
```

Arguments

`f`	array of `characters` or a `function` returning a `numeric` array.
`bounds`	`list` containing the lower and upper bounds for each variable. If the two bounds coincide, or if a single number is specified, the corresponding variable is not integrated and its value is fixed.
`params`	`list` of additional parameters passed to `f`.

coordinates	coordinate system to use. One of: cartesian, polar, spherical, cylindrical, parabolic, parabolic-cylindrical or a character vector of scale factors for each variable.
relTol	the maximum relative tolerance.
absTol	the absolute tolerance.
method	the method to use. One of "mc", "hcubature", "pcubature", "cuhre", "divonne", "suave" or "vegas". Methods other than "mc" (naive Monte Carlo) require the **cubature** package to be installed (efficient integration in C). The defaul uses "hcubature" if **cubature** is installed or "mc" otherwise.
vectorize	logical. Use vectorization? If TRUE, it can significantly boost performance but f needs to handle the vector of inputs appropriately. The default uses FALSE if f is a function, TRUE otherwise.
drop	if TRUE, return the integral as a vector and not as an array for vector-valued functions.
verbose	logical. Print on progress?
...	additional arguments passed to cubintegrate, when method "hcubature", "pcubature", "cuhre", "divonne", "suave" or "vegas" is used.

Details

The function integrates seamlessly with **cubature** for efficient numerical integration in C. If the package **cubature** is not installed, the function implements a naive Monte Carlo integration by default. For arbitrary orthogonal coordinates $q_1 \dots q_n$ the integral is computed as:

$$\int J \cdot f(q_1 \dots q_n) dq_1 \dots dq_n$$

where $J = \prod_i h_i$ is the Jacobian determinant of the transformation and is equal to the product of the scale factors $h_1 \dots h_n$.

Value

list with components

value the final estimate of the integral.

error estimate of the modulus of the absolute error.

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

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other integrals: ode()

Examples

```
### unidimensional integral
i <- integral("sin(x)", bounds = list(x = c(0,pi)))
i$value

### multidimensional integral
f <- function(x,y) x*y
i <- integral(f, bounds = list(x = c(0,1), y = c(0,1)))
i$value

### vector-valued integrals
f <- function(x,y) c(x, y, x*y)
i <- integral(f, bounds = list(x = c(0,1), y = c(0,1)))
i$value

### tensor-valued integrals
f <- function(x,y) array(c(x^2, x*y, x*y, y^2), dim = c(2,2))
i <- integral(f, bounds = list(x = c(0,1), y = c(0,1)))
i$value

### area of a circle
i <- integral(1,
              bounds = list(r = c(0,1), theta = c(0,2*pi)),
              coordinates = "polar")
i$value

### surface of a sphere
i <- integral(1,
              bounds = list(r = 1, theta = c(0,pi), phi = c(0,2*pi)),
              coordinates = "spherical")
i$value

### volume of a sphere
i <- integral(1,
         bounds = list(r = c(0,1), theta = c(0,pi), phi = c(0,2*pi)),
         coordinates = "spherical")
i$value
```

`jacobian` *Numerical and Symbolic Jacobian*

Description

Computes the numerical Jacobian of `functions` or the symbolic Jacobian of `characters` in arbitrary orthogonal coordinate systems.

Usage

```
jacobian(
  f,
  var,
  params = list(),
  coordinates = "cartesian",
  accuracy = 4,
  stepsize = NULL
)

f %jacobian% var
```

Arguments

f	array of characters or a function returning a numeric array.
var	vector giving the variable names with respect to which the derivatives are to be computed and/or the point where the derivatives are to be evaluated. See derivative.
params	list of additional parameters passed to f.
coordinates	coordinate system to use. One of: cartesian, polar, spherical, cylindrical, parabolic, parabolic-cylindrical or a vector of scale factors for each variable.
accuracy	degree of accuracy for numerical derivatives.
stepsize	finite differences stepsize for numerical derivatives. It is based on the precision of the machine by default.

Details

The function is basically a wrapper for gradient with drop=FALSE.

Value

array.

Functions

- f %jacobian% var: binary operator with default parameters.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other differential operators: curl(), derivative(), divergence(), gradient(), hessian(), laplacian()

Examples

```
### symbolic Jacobian
jacobian("x*y*z", var = c("x", "y", "z"))

### numerical Jacobian in (x=1, y=2, z=3)
f <- function(x, y, z) x*y*z
jacobian(f = f, var = c(x=1, y=2, z=3))

### vectorized interface
f <- function(x) x[1]*x[2]*x[3]
jacobian(f = f, var = c(1, 2, 3))

### symbolic vector-valued functions
f <- c("y*sin(x)", "x*cos(y)")
jacobian(f = f, var = c("x","y"))

### numerical vector-valued functions
f <- function(x) c(sum(x), prod(x))
jacobian(f = f, var = c(0,0,0))

### binary operator
"x*y^2" %jacobian% c(x=1, y=3)
```

`laplacian` *Numerical and Symbolic Laplacian*

Description

Computes the numerical Laplacian of `functions` or the symbolic Laplacian of `characters` in arbitrary orthogonal coordinate systems.

Usage

```
laplacian(
  f,
  var,
  params = list(),
  coordinates = "cartesian",
  accuracy = 4,
  stepsize = NULL,
  drop = TRUE
)

f %laplacian% var
```

Arguments

f	array of characters or a function returning a numeric array.
var	vector giving the variable names with respect to which the derivatives are to be computed and/or the point where the derivatives are to be evaluated. See derivative.
params	list of additional parameters passed to f.
coordinates	coordinate system to use. One of: cartesian, polar, spherical, cylindrical, parabolic, parabolic-cylindrical or a vector of scale factors for each variable.
accuracy	degree of accuracy for numerical derivatives.
stepsize	finite differences stepsize for numerical derivatives. It is based on the precision of the machine by default.
drop	if TRUE, return the Laplacian as a scalar and not as an array for scalar-valued functions.

Details

The Laplacian is a differential operator given by the divergence of the gradient of a scalar-valued function F, resulting in a scalar value giving the flux density of the gradient flow of a function. The laplacian is computed in arbitrary orthogonal coordinate systems using the scale factors h_i:

$$\nabla^2 F = \frac{1}{J}\sum_i \partial_i \left(\frac{J}{h_i^2} \partial_i F \right)$$

where $J = \prod_i h_i$. When the function F is a tensor-valued function $F_{d_1 \ldots d_n}$, the laplacian is computed for each scalar component:

$$(\nabla^2 F)_{d_1 \ldots d_n} = \frac{1}{J}\sum_i \partial_i \left(\frac{J}{h_i^2} \partial_i F_{d_1 \ldots d_n} \right)$$

Value

Scalar for scalar-valued functions when drop=TRUE, array otherwise.

Functions

- f %laplacian% var: binary operator with default parameters.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other differential operators: curl(), derivative(), divergence(), gradient(), hessian(), jacobian()

Examples

```
### symbolic Laplacian
laplacian("x^3+y^3+z^3", var = c("x","y","z"))

### numerical Laplacian in (x=1, y=1, z=1)
f <- function(x, y, z) x^3+y^3+z^3
laplacian(f = f, var = c(x=1, y=1, z=1))

### vectorized interface
f <- function(x) sum(x^3)
laplacian(f = f, var = c(1, 1, 1))

### symbolic vector-valued functions
f <- array(c("x^2","x*y","x*y","y^2"), dim = c(2,2))
laplacian(f = f, var = c("x","y"))

### numerical vector-valued functions
f <- function(x, y) array(c(x^2,x*y,x*y,y^2), dim = c(2,2))
laplacian(f = f, var = c(x=0,y=0))

### binary operator
"x^3+y^3+z^3" %laplacian% c("x","y","z")
```

---

mx *Numerical and Symbolic Matrix Product*

Description

Multiplies two numeric or character 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

```
mx(x, y)

x %mx% y
```

Arguments

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

Value

matrix.

Functions

- x %mx% y: binary operator.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other matrix algebra: mxdet(), mxinv(), mxtr()

Examples

```
### numeric inner product
x <- 1:4
mx(x, x)

### symbolic inner product
x <- letters[1:4]
mx(x, x)

### numeric matrix product
x <- letters[1:4]
y <- diag(4)
mx(x, y)

### symbolic matrix product
x <- array(1:12, dim = c(3,4))
y <- letters[1:4]
mx(x, y)

### binary operator
x <- array(1:12, dim = c(3,4))
y <- letters[1:4]
x %mx% y
```

---

mxdet *Numerical and Symbolic Determinant*

Description

Computes the determinant of a numeric or character matrix.

Usage

```
mxdet(x)
```

Arguments

x	numeric or character matrix.

Value

numeric or character.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other matrix algebra: mxinv(), mxtr(), mx()

Examples

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

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

---

mxinv *Numerical and Symbolic Matrix Inverse*

Description

Computes the inverse of a numeric or character matrix.

Usage

```
mxinv(x)
```

Arguments

x	numeric or character matrix.

Value

numeric or character matrix.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other matrix algebra: `mxdet()`, `mxtr()`, `mx()`

Examples

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

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

`mxtr` *Numerical and Symbolic Matrix Trace*

Description

Computes the trace of a `numeric` or `character` matrix.

Usage

```
mxtr(x)
```

Arguments

`x`	`numeric` or `character` matrix.

Value

`numeric` or `character`.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other matrix algebra: `mxdet()`, `mxinv()`, `mx()`

Examples

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

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

---

ode *Ordinary Differential Equations*

Description

Solves a numerical or symbolic system of ordinary differential equations.

Usage

```
ode(
  f,
  var,
  times,
  timevar = NULL,
  params = list(),
  method = "rk4",
  drop = FALSE
)
```

Arguments

f	vector of characters, or a function returning a numeric vector, giving the values of the derivatives in the ODE system at time timevar. See examples.
var	vector giving the initial conditions. See examples.
times	discretization sequence, the first value represents the initial time.
timevar	the time variable used by f, if any.
params	list of additional parameters passed to f.
method	the solver to use. One of "rk4" (Runge-Kutta) or "euler" (Euler).
drop	if TRUE, return only the final solution instead of the whole trajectory.

Value

Vector of final solutions if drop=TRUE, otherwise a matrix with as many rows as elements in times and as many columns as elements in var.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other integrals: `integral()`

Examples

```
## ================================================
## Example: symbolic system
## System:  dx = x dt
## Initial: x0 = 1
## ================================================
f <- "x"
var <- c(x=1)
times <- seq(0, 2*pi, by=0.001)
x <- ode(f, var, times)
plot(times, x, type = "l")

## ================================================
## Example: time dependent system
## System:  dx = cos(t) dt
## Initial: x0 = 0
## ================================================
f <- "cos(t)"
var <- c(x=0)
times <- seq(0, 2*pi, by=0.001)
x <- ode(f, var, times, timevar = "t")
plot(times, x, type = "l")

## ================================================
## Example: multivariate time dependent system
## System:  dx = x dt
##          dy = x*(1+cos(10*t)) dt
## Initial: x0 = 1
##          y0 = 1
## ================================================
f <- c("x", "x*(1+cos(10*t))")
var <- c(x=1, y=1)
times <- seq(0, 2*pi, by=0.001)
x <- ode(f, var, times, timevar = "t")
matplot(times, x, type = "l", lty = 1, col = 1:2)

## ================================================
## Example: numerical system
## System:  dx = x dt
##          dy = y dt
## Initial: x0 = 1
##          y0 = 2
## ================================================
```

```
f <- function(x, y) c(x, y)
var <- c(x=1, y=2)
times <- seq(0, 2*pi, by=0.001)
x <- ode(f, var, times)
matplot(times, x, type = "l", lty = 1, col = 1:2)

## ================================================
## Example: vectorized interface
## System:  dx = x dt
##          dy = y dt
##          dz = y*(1+cos(10*t)) dt
## Initial: x0 = 1
##          y0 = 2
##          z0 = 2
## ================================================
f <- function(x, t) c(x[1], x[2], x[2]*(1+cos(10*t)))
var <- c(1,2,2)
times <- seq(0, 2*pi, by=0.001)
x <- ode(f, var, times, timevar = "t")
matplot(times, x, type = "l", lty = 1, col = 1:3)
```

`partitions` *Integer Partitions*

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 `matrix` if `length>0` and `fill=TRUE`.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

Examples

```
### partitions of 4
partitions(4)

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

### partitions of 4 with max element equal to 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 2 elements, fill with zeros and permute
partitions(4, length = 2, 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 Expansion*

Description

Computes the Taylor series of `functions` or `characters`.

Usage

```
taylor(
  f,
  var,
  params = list(),
  order = 1,
  accuracy = 4,
  stepsize = NULL,
  zero = 1e-07
)
```

Arguments

f	character, or function returning a numeric scalar value.
var	vector giving the variable names with respect to which the derivatives are to be computed and/or the point where the derivatives are to be evaluated (the center of the Taylor series). See derivative.
params	list of additional parameters passed to f.
order	the order of the Taylor approximation.
accuracy	degree of accuracy for numerical derivatives.
stepsize	finite differences stepsize for numerical derivatives. It is based on the precision of the machine by default.
zero	tolerance used for deciding which derivatives are zero. Absolute values less than this number are set to zero.

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.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other polynomials: hermite()

Other derivatives: derivative()

Examples

```
### univariate taylor series (in x=0)
taylor("exp(x)", var = "x", order = 2)

### univariate taylor series of user-defined functions (in x=0)
f <- function(x) exp(x)
taylor(f = f, var = c(x=0), order = 2)

### multivariate taylor series (in x=0 and y=1)
taylor("x*(y-1)", var = c(x=0, y=1), order = 4)

### multivariate taylor series of user-defined functions (in x=0 and y=1)
f <- function(x,y) x*(y-1)
taylor(f, var = c(x=0, y=1), order = 4)
```

```
### vectorized interface
f <- function(x) prod(x)
taylor(f, var = c(0,0,0), order = 3)
```

---

wrap *Wrap Characters in Parentheses*

Description

Wraps characters in round brackets.

Usage

```
wrap(x)
```

Arguments

x	character.

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

character.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other utilities: c2e(), e2c(), evaluate()

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

Elementwise difference of `numeric` or `character` arrays.

Usage

```
x %diff% y
```

Arguments

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

Value

`array`.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other basic arithmetic: `%div%()`, `%dot%()`, `%inner%()`, `%kronecker%()`, `%outer%()`, `%prod%()`, `%sum%()`

Examples

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

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

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

`%div%` *Numerical and Symbolic Division*

Description

Elementwise division of `numeric` or `character` arrays.

Usage

```
x %div% y
```

Arguments

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

Value

`array`.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other basic arithmetic: `%diff%()`, `%dot%()`, `%inner%()`, `%kronecker%()`, `%outer%()`, `%prod%()`, `%sum%()`

Examples

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

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

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

```
x %div% y
```

`%dot%` *Numerical and Symbolic Dot Product*

Description

The dot product between arrays with different dimensions is computed by taking the inner product on the last dimensions of the two arrays.

Usage

```
x %dot% y
```

Arguments

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

Details

The dot product between two arrays `A` and `B` is computed as:

$$C_{i_1 \ldots i_m} = \sum_{j_1 \ldots j_n} A_{i_1 \ldots i_m j_1 \ldots j_n} B_{j_1 \ldots j_n}$$

Value

`array`.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other basic arithmetic: `%diff%()`, `%div%()`, `%inner%()`, `%kronecker%()`, `%outer%()`, `%prod%()`, `%sum%()`

Examples

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

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

`%inner%` *Numerical and Symbolic Inner Product*

Description

Computes the inner product of two `numeric` or `character` arrays.

Usage

```
x %inner% y
```

Arguments

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

Details

The inner product between two arrays `A` and `B` is computed as:

$$C = \sum_{j_1 \ldots j_n} A_{j_1 \ldots j_n} B_{j_1 \ldots j_n}$$

Value

`numeric` or `character`.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other basic arithmetic: `%diff%()`, `%div%()`, `%dot%()`, `%kronecker%()`, `%outer%()`, `%prod%()`, `%sum%()`

Examples

```
### numeric inner product
x <- array(1:4, dim = c(2,2))
x %inner% x

### symbolic inner product
x <- array(letters[1:4], dim = c(2,2))
x %inner% x
```

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

Description

Computes the generalised Kronecker product of two `numeric` or `character` arrays.

Usage

```
x %kronecker% y
```

Arguments

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

Value

`array`.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other basic arithmetic: `%diff%()`, `%div%()`, `%dot%()`, `%inner%()`, `%outer%()`, `%prod%()`, `%sum%()`

Examples

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

### symbolic Kronecker product
array(1:4, dim = c(2,2)) %kronecker% c("a","b")
```

`%outer%` *Numerical and Symbolic Outer Product*

Description

Computes the outer product of two `numeric` or `character` arrays.

Usage

```
x %outer% y
```

Arguments

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

Details

The outer product between two arrays `A` and `B` is computed as:

$$C_{i_1 \ldots i_m j_1 \ldots j_n} = A_{i_1 \ldots i_m} B_{j_1 \ldots j_n}$$

Value

`array`.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other basic arithmetic: `%diff%()`, `%div%()`, `%dot%()`, `%inner%()`, `%kronecker%()`, `%prod%()`, `%sum%()`

Examples

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

### symbolic outer product
c("a","b") %outer% c("c","d")
```

`%prod%` *Numerical and Symbolic Product*

Description

Elementwise product of `numeric` or `character` arrays.

Usage

```
x %prod% y
```

Arguments

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

Value

`array`.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other basic arithmetic: `%diff%()`, `%div%()`, `%dot%()`, `%inner%()`, `%kronecker%()`, `%outer%()`, `%sum%()`

Examples

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

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

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

`%sum%` *Numerical and Symbolic Sum*

Description

Elementwise sum of `numeric` or `character` arrays.

Usage

```
x %sum% y
```

Arguments

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

Value

`array`.

References

Guidotti, E. (2020). "calculus: High dimensional numerical and symbolic calculus in R". https://arxiv.org/abs/2101.00086

See Also

Other basic arithmetic: `%diff%()`, `%div%()`, `%dot%()`, `%inner%()`, `%kronecker%()`, `%outer%()`, `%prod%()`

Examples

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

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

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

Index