Package: mvp (via r-universe)

November 1, 2024

Type Package

Title Fast Symbolic Multivariate Polynomials

Version 1.0-18

Depends R (>= 3.5.0), methods

Suggests knitr, rmarkdown, microbenchmark, testthat, spray, magrittr, covr

VignetteBuilder knitr

Maintainer Robin K. S. Hankin <hankin.robin@gmail.com>

Description Fast manipulation of symbolic multivariate polynomials using the 'Map' class of the Standard Template Library. The package uses print and coercion methods from the 'mpoly' package but offers speed improvements. It is comparable in speed to the 'spray' package for sparse arrays, but retains the symbolic benefits of 'mpoly'. To cite the package in publications, use Hankin 2022 <doi:10.48550/ARXIV.2210.15991>. Uses 'disordR' discipline.

License GPL (>= 2)

Imports Rcpp (>= 1.0-7), partitions, magic, digest, disordR (>= 0.9-7), numbers, mpoly (>= 1.1.0), mathjaxr

LinkingTo Rcpp

URL https://github.com/RobinHankin/mvp, https://robinhankin.github.io/mvp/

BugReports https://github.com/RobinHankin/mvp/issues

RdMacros mathjaxr

LazyData FALSE

Repository https://robinhankin.r-universe.dev

RemoteUrl https://github.com/robinhankin/mvp

RemoteRef HEAD

RemoteSha ffad10cd165435e1d916a6a7fe43e056bd454665

Contents

mvp-package . . . 2
allvars . . . 3
as.function.mvp . . . 4
coeffs . . . 4
constant . . . 7
deriv . . . 8
drop . . . 9
horner . . . 10
invert . . . 11
kahle . . . 12
knight . . . 13
letters . . . 14
lowlevel . . . 15
mpoly . . . 16
mvp . . . 16
ooom . . . 18
Ops.mvp . . . 19
print . . . 20
rmvp . . . 21
series . . . 22
special . . . 24
subs . . . 25
summary . . . 27
zero . . . 28

Index **30**

`mvp-package` *Fast Symbolic Multivariate Polynomials*

Description

Fast manipulation of symbolic multivariate polynomials using the 'Map' class of the Standard Template Library. The package uses print and coercion methods from the 'mpoly' package but offers speed improvements. It is comparable in speed to the 'spray' package for sparse arrays, but retains the symbolic benefits of 'mpoly'. To cite the package in publications, use Hankin 2022 <doi:10.48550/ARXIV.2210.15991>. Uses 'disordR' discipline.

Details

The DESCRIPTION file: This package was not yet installed at build time.

Index: This package was not yet installed at build time.

Author(s)

Robin K. S. Hankin [aut, cre] (<https://orcid.org/0000-0001-5982-0415>)

Maintainer: Robin K. S. Hankin <hankin.robin@gmail.com>

Examples

```
(p <- as.mvp("1+x+x*y+x^5"))

p + as.mvp("a+b^6")

p^3

subs(p^4,x="a+b^2")
aderiv(p^2,x=4)
horner(p,1:3)
```

---

allvars *All variables in a multivariate polynomial*

Description

Returns a character vector containing all the variables present in a `mvp` object.

Usage

```
allvars(x)
```

Arguments

`x`	object of class `mvp`

Note

The character vector returned is not in any particular order

Author(s)

Robin K. S. Hankin

Examples

```
p <- rmvp(5)
p
allvars(p)
```

as.function.mvp *Functional form for multivariate polynomials*

Description

Coerces a multivariate polynomial into a function

Usage

```
## S3 method for class 'mvp'
as.function(x, ...)
```

Arguments

x	Multivariate polynomial
...	Further arguments (currently ignored)

Author(s)

Robin K. S. Hankin

Examples

```
p <- as.mvp("1+a^2 + a*b^2 + c")
p
f <- as.function(p)
f

f(a=1)
f(a=1,b=2)
f(a=1,b=2,c=3)             # coerces to a scalar
f(a=1,b=2,c=3,drop=FALSE)  # formal mvp object
```

coeffs *Functionality for* coeffs *objects*

Description

Function coeffs() allows arithmetic operators to be used for the coefficients of multivariate polynomials, bearing in mind that the order of coefficients is not determined. It uses the disord class of the **disordR** package.

"Pure" extraction and replacement (as in a[i] and a[i] <- value is implemented experimentally. The code for extraction is cute but not particularly efficient.

Usage

```
coeffs(x)
vars(x)
powers(x)
coeffs(x) <- value
```

Arguments

x	Object of class disord
value	Object of class disord, or length-1 numeric vector

Details

(much of the discussion below appears in the vignette of the **disordR** package).

Accessing elements of an mvp object is problematic because the order of the terms of an mvp object is not well-defined. This is because the map class of the STL does not specify an order for the key-value pairs (and indeed the actual order in which they are stored may be implementation dependent). The situation is similar to the hyper2 package which uses the STL in a similar way.

A coeffs object is a vector of coefficients of a mvp object. But it is not a conventional vector; in a conventional vector, we can identify the first element unambiguously, and the second, and so on. An mvp is a map from terms to coefficients, and a map has no intrinsic ordering: the maps

```
{x -> 1, y -> 3, xy^3 -> 4}
```

and

```
{xy^3 -> 4, x -> 1, y -> 3}
```

are the same map and correspond to the same multinomial (symbolically, $x + 3y + 4xy^3 = 4xy^3 + x + 3y$). Thus the coefficients of the multinomial might be c(1,3,4) or c(4,1,3), or indeed any ordering. But note that any particular ordering imposes an ordering on the terms. If we choose c(1,3,4) then the terms are x,y,xy^3, and if we choose c(4,1,3) the terms are xy^3,x,y.

In the package, coeffs() returns an object of class disord. This class of object has a slot for the coefficients in the form of a numeric R vector, but also another slot which uses hash codes to prevent users from misusing the ordering of the numeric vector.

For example, a multinomial x+2y+3z might have coefficients c(1,2,3) or c(3,1,2). Package idiom to extract the coefficients of a multivariate polynomial a is coeffs(a); but this cannot return a standard numeric vector because a numeric vector has elements in a particular order, and the coefficients of a multivariate polynomial are stored in an implementation-specific (and thus unknown) order.

Suppose we have two multivariate polynomials, a as defined as above with a=x+2y+3z and b=x+3y+4z. Even though a+b is well-defined algebraically, and coeffs(a+b) will return a well-defined mvp_coeffs object, idiom such as coeffs(a) + coeffs(b) is not defined because there is no guarantee that the coefficients of the two multivariate polynomials are stored in the same order. We might have c(1,2,3)+c(1,3,4)=c(2,5,7) or c(1,2,3)+c(1,4,3)=c(2,6,6), with neither being more "correct" than the other. In the package, coeffs(a) + coeffs(b) will return an error. In the same

way `coeffs(a) + 1:3` is not defined and will return an error. Further, idiom such as `coeffs(a) <- 1:3` and `coeffs(a) <- coeffs(b)` are not defined and will return an error. However, note that `coeffs(a) + coeffs(a)` and `coeffs(a)+coeffs(a)^2` are fine, these returning a `mvp_coeffs` object specific to `a`.

Idiom such as `coeffs(a) <- coeffs(a)^2` is fine too, for one does not need to know the order of the coefficients on either side, so long as the order is the same on both sides. That would translate into idiomatic English: "the coefficient of each term of `a` becomes its square"; note that this operation is insensitive to the order of coefficients. The whole shebang is intended to make idiom such as `coeffs(a) <- coeffs(a)%%2` possible (so we can manipulate polynomials over finite rings, here $Z/2Z$).

The replacement methods are defined so that an expression like `coeffs(a)[coeffs(a) > 5] <- 5` works as expected; the English idiom would be "Replace any coefficient greater than 5 with 5".

To fix ideas, consider `a <- rmvp(8)`. Extraction presents issues; consider `coeffs(a)<5`. This object has Boolean elements but has the same ordering ambiguity as `coeffs(a)`. One might expect that we could use this to extract elements of `coeffs(a)`, specifically elements less than 5. However, `coeffs(a)[coeffs(a)<5]` in isolation is meaningless: what can be done with such an object? However, it makes sense on the left hand side of an assignment, as long as the right hand side is a length-one vector. Idiom such as

- `coeffs(a)[coeffs(a)<5] <- 4+coeffs(a)[coeffs(a)<5]`
- `coeffs(a) <- pmax(a,3)`

is algebraically meaningful ("Add 4 to any element less than 5"; "coefficients become the pairwise maximum of themselves and 3"). The **disordR** package uses `pmaxdis()` rather than `pmax()` for technical reasons.

So the output of `coeffs(x)` is defined only up to an unknown rearrangement. The same considerations apply to the output of `vars()`, which returns a list of character vectors in an undefined order, and the output of `powers()`, which returns a numeric list whose elements are in an undefined order. However, even though the order of these three objects is undefined individually, their ordering is jointly consistent in the sense that the first element of `coeffs(x)` corresponds to the first element of `vars(x)` and the first element of `powers(x)`. The identity of this element is not defined—but whatever it is, the first element of all three accessor methods refers to it.

Note also that a single term (something like `4a^3*b*c^6`) has the same issue: the variables are not stored in a well-defined order. This does not matter because the algebraic value of the term does not depend on the order in which the variables appear and this term would be equivalent to `4b*c^6*a^3`.

Author(s)

Robin K. S. Hankin

Examples

```
(x <- 5+rmvp(6))
(y <- 2+rmvp(6))

coeffs(x)^2
coeffs(y) <- coeffs(y)%%3  # fine, all coeffs of y now modulo 3
y
```

```
coeffs(y) <- 4
y

## Not run:
coeffs(x) <- coeffs(y)          # not defined, will give an error
coeffs(x) <- seq_len(nterms(x)) # not defined, will give an error

## End(Not run)
```

`constant` *The constant term*

Description

Get and set the constant term of an `mvp` object

Usage

```
## S3 method for class 'mvp'
constant(x)
## S3 replacement method for class 'mvp'
constant(x) <- value
## S3 method for class 'numeric'
constant(x)
is.constant(x)
```

Arguments

`x`	Object of class `mvp`
`value`	Scalar value for the constant

Details

The constant term in a polynomial is the coefficient of the empty term. In an `mvp` object, the map `{} -> c`, implies that `c` is the constant.

If `x` is an `mvp` object, `constant(x)` returns the value of the constant in the multivariate polynomial; if `x` is numeric, it returns a constant multivariate polynomial with value `x`.

Function `is.constant()` returns `TRUE` if its argument has no variables and `FALSE` otherwise.

Author(s)

Robin K. S. Hankin

Examples

```
a <- rmvp(5)+4
a
constant(a)
constant(a) <- 33
a

constant(0)  # the zero mvp
```

---

deriv *Differentiation of* mvp *objects*

Description

Differentiation of mvp objects

Usage

```
## S3 method for class 'mvp'
deriv(expr, v, ...)
## S3 method for class 'mvp'
aderiv(expr, ...)
```

Arguments

expr	Object of class mvp
v	Character vector. Elements denote variables to differentiate with respect to
...	Further arguments. In deriv(), a non-negative integer argument specifies the order of the differential, and in aderiv() the argument names specify the differentials and their values their respective orders

Details

deriv(S,v) returns $\frac{\partial^r S}{\partial v_1 \partial v_2 ... \partial v_r}$. Here, v is a (character) vector of symbols.

deriv(S,v,n) returns the n-th derivative of S with respect to symbol v, $\frac{\partial^n S}{\partial v^n}$.

aderiv() uses the ellipsis construction with the names of the argument being the variable to be differentiated with respect to. Thus aderiv(S,x=1,y=2) returns $\frac{\partial^3 S}{\partial x \partial y^2}$.

Author(s)

Robin K. S. Hankin

See Also

taylor

Examples

```
(p <- rmvp(10,4,15,5))
deriv(p,"a")
deriv(p,"a",3)
deriv(p,letters[1:3])
deriv(p,rev(letters[1:3]))  # should be the same

aderiv(p,a=1,b=2,c=1)

## verify the chain rule:
x <- rmvp(7,symbols=6)
v <- allvars(x)[1]
s <- as.mvp("1  +  y  -  y^2 zz  +  y^3 z^2")
LHS <- subsmvp(deriv(x,v)*deriv(s,"y"),v,s)   # dx/ds*ds/dy
RHS <- deriv(subsmvp(x,v,s),"y")              # dx/dy

LHS - RHS # should be zero
```

---

drop *Drop empty variables*

Description

Convert an mvp object which is a pure constant into a scalar whose value is the coefficient of the empty term.

A few functions in the package (currently subs(), subsy()) take a drop argument that behaves much like the drop argument in base extraction.

Function drop() is an S4 generic, which is why the package calls setOldClass().

Function drop() was formerly called lose().

Usage

```
## S4 method for signature 'mvp'
drop(x)
```

Arguments

x	Object of class mvp

Author(s)

Robin K. S. Hankin

See Also

subs

Examples

```
(m1 <- as.mvp("1+bish +bash^2 + bosh^3"))
(m2 <- as.mvp("bish +bash^2 + bosh^3"))

m1-m2         # an mvp object
drop(m1-m2)   # numeric
```

`horner` *Horner's method*

Description

Horner's method for multivariate polynomials

Usage

```
horner(P,v)
```

Arguments

`P`	Multivariate polynomial
`v`	Numeric vector of coefficients

Details

Given a polynomial

$$p(x) = a_0 + a_1 + a_2x^2 + \cdots + a_nx^n$$

it is possible to express $p(x)$ in the algebraically equivalent form

$$p(x) = a_0 + x\left(a_1 + x\left(a_2 + \cdots + x\left(a_{n-1} + xa_n\right)\cdots\right)\right)$$

which is much more efficient for evaluation, as it requires only n multiplications and n additions, and this is optimal. But this is not implemented here because it's efficient. It is implemented because it works if x is itself a (multivariate) polynomial, and that is the second coolest thing ever. The coolest thing ever is the `Reduce()` function.

Author(s)

Robin K. S. Hankin

See Also

`ooom`

Examples

```
horner("x",1:5)
horner("x+y",1:3)

w <- as.mvp("x+y^2")
stopifnot(1 + 2*w + 3*w^2 == horner(w,1:3))  # note off-by-one issue

"x+y+x*y" |> horner(1:3) |> horner(1:2)
```

`invert` *Replace symbols with their reciprocals*

Description

Given an `mvp` object, replace one or more symbols with their reciprocals

Usage

```
invert(p, v)
```

Arguments

`p`	Object (coerced to) `mvp` form
`v`	Character vector of symbols to be replaced with their reciprocal; missing interpreted as replace all symbols

Author(s)

Robin K. S. Hankin

See Also

`subs`

Examples

```
invert("x")

(P <- as.mvp("1+a+6*a^2 -7*a*b"))
invert(P,"a")
```

kahle *A sparse multivariate polynomial*

Description

A sparse multivariate polynomial inspired by Kahle (2013)

Usage

```
kahle(n = 26, r = 1, p = 1, coeffs = 1, symbols = letters)
```

Arguments

n	Number of different symbols to use
r	Number of symbols in a single term
p	Power of each symbol in each terms
coeffs	Coefficients of the terms
symbols	Alphabet of symbols

Author(s)

Robin K. S. Hankin

References

David Kahle 2013. "**mpoly**: multivariate polynomials in R". *R Journal*, volume 5/1.

See Also

special

Examples

```
kahle()  # a+b+...+z
kahle(r=2,p=1:2)  # Kahle's original example

## example where mvp runs faster than spray (mvp does not need a 200x200 matrix):
k <- kahle(200,r=3,p=1:3,symbols=paste("x",sprintf("%02d",1:200),sep=""))
system.time(ignore <- k^2)
#system.time(ignore <- mvp_to_spray(k)^2)   # needs spray package loaded
```

knight *Chess knight*

Description

Generating function for a chess knight on an infinite d-dimensional chessboard

Usage

```
knight(d, can_stay_still = FALSE)
```

Arguments

d	Dimension of the board
can_stay_still	Boolean, with default FALSE meaning that the knight is obliged to move and FALSE meaning that it has the option of remaining on its square

Note

The function is a slight modification of `spray::knight()`.

Author(s)

Robin K. S. Hankin

Examples

```
knight(2)      # regular chess knight on a regular chess board
knight(2,TRUE) # regular chess knight that can stay still

# Q: how many ways are there for a 4D knight to return to its starting
# square after four moves?

# A:
constant(knight(4)^4)

# Q ...and how many ways in four moves or fewer?

# A1:
constant(knight(4,TRUE)^4)

# A2:
constant((1+knight(4))^4)
```

letters *Single-letter symbols*

Description

Variables a, b,... z are given their mvp semantic meaning.

Usage

```
data(lettersymbols)
```

Details

Twenty-six variables a-z are defined as their mvp semantic equivalent:

```
a <- as.mvp("a")
...
z <- as.mvp("z")
```

These objects can be generated by running script inst/symb.Rmd, which includes some further discussion and technical documentation and creates file lettersymbols.rda which resides in the data/ directory.

Letters c, q, and t might pose difficulties.

Author(s)

Robin K. S. Hankin

Examples

```
data(lettersymbols)
(a+b)*(a-b)

(x + y + z)^3 - 3*(x + y + z)*(x*y + x*z + y*z) + 3*x*y*z
```

lowlevel *Low level functions*

Description

Various low-level functions that call the C routines

Usage

```
mvp_substitute(allnames,allpowers,coefficients,v,values)
mvp_substitute_mvp(allnames1, allpowers1, coefficients1, allnames2, allpowers2,
    coefficients2, v)
mvp_vectorised_substitute(allnames, allpowers, coefficients, M, nrows, ncols, v)
mvp_prod(allnames1,allpowers1,coefficients1,allnames2,allpowers2,coefficients2)
mvp_add(allnames1, allpowers1, coefficients1, allnames2, allpowers2,coefficients2)
simplify(allnames,allpowers,coefficients)
mvp_deriv(allnames, allpowers, coefficients, v)
mvp_power(allnames, allpowers, coefficients, n)
```

Arguments

allnames, allpowers, coefficients, allnames1, allpowers1, coefficients1, allnames2, allpowers2, coefficients2, v, values, n, M, nrows, ncols
: Variables sent to the C routines

Details

These functions call the functions defined in RcppExports.R

Note

These functions are not intended for the end-user. Use the syntactic sugar (as in a+b or a*b or a^n), or functions like mvp_plus_mvp(), which are more user-friendly.

Author(s)

Robin K. S. Hankin

mpoly *Conversion to and from mpoly form*

Description

The **mpoly** package by David Kahle provides similar functionality to this package, and the functions documented here convert between mpoly and `mvp` objects. The mvp package uses `mpoly::mp()` to convert character strings to `mvp` objects.

Usage

```
mpoly_to_mvp(m)
## S3 method for class 'mvp'
as.mpoly(x,...)
```

Arguments

`m`	object of class mvp
`x`	object of class mpoly
`...`	further arguments, currently ignored

Author(s)

Robin K. S. Hankin

Examples

```
x <- rmvp(5)

x == mpoly_to_mvp(mpoly::as.mpoly(x))        # should be TRUE
```

mvp *Multivariate polynomials, mvp objects*

Description

Create, test for, and coerce to, `mvp` objects

Usage

```
mvp(vars, powers, coeffs)
is_ok_mvp(vars,powers,coeffs)
is.mvp(x)
as.mvp(x)
## S3 method for class 'character'
as.mvp(x)
## S3 method for class 'list'
as.mvp(x)
## S3 method for class 'mpoly'
as.mvp(x)
## S3 method for class 'mvp'
as.mvp(x)
## S3 method for class 'numeric'
as.mvp(x)
```

Arguments

vars	List of variables comprising each term of an mvp object
powers	List of powers corresponding to the variables of the vars argument
coeffs	Numeric vector corresponding to the coefficients to each element of the var and powers lists
x	Object to be coerced to or tested for being class mvp

Details

Function mvp() is the formal creation mechanism for mvp objects. However, it is not very user-friendly; it is better to use as.mvp() in day-to-day use.

Function is_ok_mvp() checks for consistency of its arguments.

Author(s)

Robin K. S. Hankin

Examples

```
mvp(list("x", c("x","y"), "a", c("y","x")), list(1,1:2,3,c(-1,4)), 1:4)

## Note how the terms appear in an arbitrary order, as do
## the symbols within a term.

kahle  <- mvp(
    vars   = split(cbind(letters,letters[c(26,1:25)]),rep(seq_len(26),each=2)),
    powers = rep(list(1:2),26),
    coeffs = 1:26
)
kahle
## again note arbitrary order of terms and symbols within a term
```

```
## Standard arithmetic rules apply:

a <- as.mvp("1 + 4*x*y + 7*z")
b <- as.mvp("-7*z + 3*x^34 - 2*z*x")

a+b
a*b^2

(a+b)*(a-b) == a^2-b^2 # should be TRUE


## variable "xy" is distinct from "x*y":

as.mvp("x + y + xy")^2
as.mvp(paste(state.name[1:5],collapse="+"))^2
```

---

ooom *One over one minus a multivariate polynomial*

Description

Uses Taylor's theorem to give one over one minus a multipol

Usage

```
ooom(P,n)
```

Arguments

n	Order of expansion
P	Multivariate polynomial

Author(s)

Robin K. S. Hankin

See Also

horner

Examples

```
ooom("x",5)
ooom("x",5) * as.mvp("1-x")  # 1 + O(x^6)


ooom("x+y",4)
```

```
"x+y" |> ooom(5) |> aderiv(x=2,y=1)
```

`Ops.mvp` *Arithmetic Ops Group Methods for* `mvp` *objects*

Description

Allows arithmetic operators to be used for multivariate polynomials such as addition, multiplication, integer powers, etc.

Usage

```
## S3 method for class 'mvp'
Ops(e1, e2)
mvp_negative(S)
mvp_times_mvp(S1,S2)
mvp_times_scalar(S,x)
mvp_plus_mvp(S1,S2)
mvp_plus_numeric(S,x)
mvp_eq_mvp(S1,S2)
mvp_modulo(S1,S2)
```

Arguments

`e1`, `e2`, `S`, `S1`, `S2`	Objects of class `mvp`
`x`	Scalar, length one numeric vector

Details

The function `Ops.mvp()` passes unary and binary arithmetic operators "+", "-", "*" and "^" to the appropriate specialist function.

The most interesting operator is "*", which is passed to `mvp_times_mvp()`. I guess "+" is quite interesting too.

The caret "^" denotes arithmetic exponentiation, as in `x^3==x*x*x`. As an experimental feature, this is (sort of) vectorised: if `n` is a vector, then `a^n` returns the sum of `a` raised to the power of each element of `n`. For example, `a^c(n1,n2,n3)` is `a^n1 + a^n2 + a^n3`. Internally, `n` is tabulated in the interests of efficiency, so `a^c(0,2,5,5,5) = 1 + a^2 + 3a^5` is evaluated with only a single fifth power. Similar functionality is implemented in the **freealg** package.

Value

The high-level functions documented here return an object of `mvp`, the low-level functions documented at `lowlevel.Rd` return lists. But don't use the low-level functions.

Note

Function `mvp_modulo()` is distinctly sub-optimal and `inst/mvp_modulo.Rmd` details ideas for better implementation.

Author(s)

Robin K. S. Hankin

See Also

`lowlevel`

Examples

```
(p1 <- rmvp(3))
(p2 <- rmvp(3))

p1*p2

p1+p2

p1^3


p1*(p1+p2) == p1^2+p1*p2  # should be TRUE
```

---

print *Print methods for* mvp *objects*

Description

Print methods for `mvp` objects: to print, an `mvp` object is coerced to `mpoly` form and the `mpoly` print method used.

Usage

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

Arguments

`x`	Object of class `mvp`, coerced to `mpoly` form
`...`	Further arguments

Value

Returns its argument invisibly

Author(s)

Robin K. S. Hankin

Examples

```
a <- rmvp(4)
a
print(a)
print(a,stars=TRUE)
print(a,varorder=rev(letters))
```

---

rmvp *Random multivariate polynomials*

Description

Random multivariate polynomials, intended as quick "get you going" examples of mvp objects

Usage

```
rhmvp(n=7,size=4,pow=6,symbols=6)
rmvp(n=7,size=4,pow=6,symbols=6)
rmvpp(n=30,size=9,pow=20,symbols=15)
rmvppp(n=100,size=15,pow=99,symbols)
```

Arguments

n	Number of terms to generate
size	Maximum number of symbols in each term
pow	Maximum power of each symbol
symbols	Symbols to use; if numeric, interpret as the first symbols letters of the alphabet

Details

Function rhmvp() returns a random homogeneous mvp. Function rmvp() returns a possibly nonhomogenous mvp and functions rmvpp() and rmvppp() return, by default, progressively more complicated mvp objects. Function rmvppp() returns a polynomial with multi-letter variable names.

Value

Returns a multivariate polynomial, an object of class mvp

Author(s)

Robin K. S. Hankin

Examples

```
rhmvp()
rmvp()
rmvpp()
rmvppp()
```

`series` *Decomposition of multivariate polynomials by powers*

Description

Power series of multivariate polynomials, in various forms

Usage

```
trunc(S,n)
truncall(S,n)
trunc1(S,...)
series(S,v,showsymb=TRUE)
## S3 method for class 'series'
print(x,...)
onevarpow(S,...)
taylor(S,vx,va,debug=FALSE)
mvp_taylor_onevar(allnames,allpowers,coefficients, v, n)
mvp_taylor_allvars(allnames,allpowers,coefficients, n)
mvp_taylor_onepower_onevar(allnames, allpowers, coefficients, v, n)
mvp_to_series(allnames, allpowers, coefficients, v)
```

Arguments

`S`	Object of class `mvp`
`n`	Non-negative integer specifying highest order to be retained
`v`	Variable to take Taylor series with respect to. If missing, total power of each term is used (except for `series()` where it is mandatory)
`x, ...`	Object of class `series` and further arguments, passed to the print method; in `trunc1()` a list of variables to truncate
`showsymb`	In function `series()`, Boolean, with default `TRUE` meaning to substitute variables like `x_m_foo` with `(x-foo)` for readability reasons; see the vignette for a discussion
`vx, va, debug`	In function `taylor()`, names of variables to take series with respect to; and a Boolean with default `FALSE` meaning to return the mvp and `TRUE` meaning to return the string that is passed to `eval()`
`allnames, allpowers, coefficients`	Components of `mvp` objects

Details

Function `onevarpow()` returns just the terms in which the symbols corresponding to the named arguments have powers equal to the arguments' powers. Thus:

```
 onevarpow(as.mvp("x*y*z + 3*x*y^2 + 7*x*y^2*z^6 + x*y^3"),x=1,y=2)
mvp object algebraically equal to
3  +  7 z^6
```

Above, we see that only the terms with `x^1*y^2` have been extracted, corresponding to arguments `x=1,y=2`.

Function `series()` returns a power series expansion of powers of variable `v`. The value returned is a list of three elements named `mvp`, `varpower`, and `variablename`. The first element is a list of `mvp` objects and the second is an integer vector of powers of variable `v` (element `variablename` is a character string holding the variable name, argument `v`).

Function `trunc(S,n)` returns the terms of `S` with the sum of the powers of the variables $\leq n$. Alternatively, it discards all terms with total power $> n$.

Function `trunc1()` is similar to `trunc()`. It takes a `mvp` object and an arbitrary number of named arguments, with names corresponding to variables and their values corresponding to the highest power in that variable to be retained. Thus `trunc1(S,x=2,y=4)` will discard any term with variable `x` raised to the power 3 or above, and also any term with variable `y` raised to the power 5 or above. The highest power of `x` will be 2 and the highest power of `y` will be 4.

Function `truncall(S,n)` discards any term of `S` with any variable raised to a power greater than `n`.

Function `series()` returns an object of class `series`; the print method for `series` objects is sensitive to the value of `getOption("mvp_mult_symbol")`; set this to `"*"` to get `mpoly`-compatible output.

Function `taylor()` is a convenience wrapper for `series()`.

Functions `mvp_taylor_onevar()`, `mvp_taylor_allvars()` and `mvp_to_series()` are low-level helper functions that are not intended for the user.

Author(s)

Robin K. S. Hankin

See Also

`deriv`

Examples

```
trunc(as.mvp("1+x")^6,2)

trunc(as.mvp("1+x+y")^3,2)      # discards all terms with total power>2
trunc1(as.mvp("1+x+y")^3,x=2)   # terms like y^3 are treated as constants

trunc(as.mvp("1+x+y^2")^3,3)    # discards x^2y^2 term (total power=4>3)
truncall(as.mvp("1+x+y^2")^3,3) # retains  x^2y^2 term (all vars to power 2)
```

```
onevarpow(as.mvp("1+x+x*y^2  + z*y^2*x"),x=1,y=2)

(p2 <- rmvp(10))
series(p2,"a")

# Works well with pipes:

f <- function(n){as.mvp(sub('n',n,'1+x^n*y'))}
Reduce(`*`,lapply(1:6,f)) |> series('y')
Reduce(`*`,lapply(1:6,f)) |> series('x')


(p <- horner("x+y",1:4))
onevarpow(p,x=2)   # coefficient of x^2
onevarpow(p,x=3)   # coefficient of x^3


p |> trunc(2)
p |> trunc1(x=2)
(p |> subs(x="x+dx") -p) |> trunc1(dx=2)

# Nice example of Horner's method:
(p <- as.mvp("x + y + 3*x*y"))
trunc(horner(p,1:5)*(1-p)^2,4)  # should be 1


## Third order taylor expansion of f(x)=sin(x+y) for x=1.1, about x=1:
(sinxpy <- horner("x+y",c(0,1,0,-1/6,0,+1/120,0,-1/5040,0,1/362880)))  # sin(x+y)
dx <- as.mvp("dx")
t3 <- sinxpy  + aderiv(sinxpy,x=1)*dx + aderiv(sinxpy,x=2)*dx^2/2 + aderiv(sinxpy,x=3)*dx^3/6
t3 <- t3 |> subs(x=1,dx=0.1)  # t3 = Taylor expansion of sin(y+1.1)
t3 |> subs(y=0.3)  - sin(1.4)  # numeric; should be small
```

`special` *Various functions to create simple multivariate polynomials*

Description

Various functions to create simple `mvp` objects such as single-term, homogeneous, and constant multivariate polynomials.

Usage

```
product(v,symbols=letters)
homog(d,power=1,symbols=letters)
linear(x,power=1,symbols=letters)
xyz(n,symbols=letters)
numeric_to_mvp(x)
```

Arguments

d, n	An integer; generally, the dimension or arity of the resulting mvp object
v, power	Integer vector of powers
x	Numeric vector of coefficients
symbols	Character vector for the symbols

Value

All functions documented here return a mvp object

Note

The functions here are related to their equivalents in the multipol and spray packages, but are not exactly the same.

Function constant() is documented at constant.Rd, but is listed below for convenience.

Author(s)

Robin K. S. Hankin

See Also

constant, zero

Examples

```
product(1:3)        #      a * b^2 * c^3
homog(3)            #      a + b + c
homog(3,2)          #      a^2  + a b + a c + b^2 + b c + c^2
linear(1:3)         #      1*a + 2*b + 3*c
constant(5)         #      5
xyz(5)              #      a*b*c*d*e
```

---

subs *Substitution*

Description

Substitute symbols in an mvp object for numbers or other multivariate polynomials

Usage

```
subs(S, ..., drop = TRUE)
subsy(S, ..., drop = TRUE)
subvec(S, ...)
subsmvp(S,v,X)
varchange(S,...)
varchange_formal(S,old,new)
namechanger(x,old,new)
```

Arguments

S, X	Multivariate polynomials
...	named arguments corresponding to variables to substitute
drop	Boolean with default TRUE meaning to return a scalar (the constant) in place of a constant mvp object
v	A string corresponding to the variable to substitute
old, new, x	The old and new variable names respectively; x is a character vector

Details

Function subs() substitutes variables for mvp objects, using a natural R idiom. Observe that this type of substitution is sensitive to order:

```
> p <- as.mvp("a b^2")
> subs(p,a="b",b="x")
mvp object algebraically equal to
x^3
> subs(p,b="x",a="b")  # same arguments, different order
mvp object algebraically equal to
b x^2
```

Functions subsy() and subsmvp() are lower-level functions, not really intended for the end-user. Function subsy() substitutes variables for numeric values (order matters if a variable is substituted more than once). Function subsmpv() takes a mvp object and substitutes another mvp object for a specific symbol.

Function subvec() substitutes the symbols of S with numerical values. It is vectorised in its ellipsis arguments with recycling rules and names behaviour inherited from cbind(). However, if the first element of ... is a matrix, then this is interpreted by rows, with symbol names given by the matrix column names; further arguments are ignored. Unlike subs(), this function is generally only useful if all symbols are given a value; unassigned symbols take a value of zero.

Function varchange() makes a *formal* variable substitution. It is useful because it can take non-standard variable names such as "(a-b)" or "?", and is used in taylor(). Function varchange_formal() does the same task, but takes two character vectors, old and new, which might be more convenient than passing named arguments. Remember that non-standard names might need to be quoted; also you might need to escape some characters, see the examples. Function namechanger() is a low-level helper function that uses regular expression idiom to substitute variable names.

Value

Functions subs(), subsy() and subsmvp() return a multivariate polynomial unless drop is TRUE in which case a length one numeric vector is returned. Function subvec() returns a numeric vector (sic! the output inherits its order from the arguments).

Author(s)

Robin K. S. Hankin

See Also

drop

Examples

```
p <- rmvp(6,2,2,letters[1:3])
p
subs(p,a=1)
subs(p,a=1,b=2)

subs(p,a="1+b x^3",b="1-y")
subs(p,a=1,b=2,c=3,drop=FALSE)

do.call(subs,c(list(as.mvp("z")),rep(c(z="C+z^2"),5)))

subvec(p,a=1,b=2,c=1:5)   # supply a named list of vectors

M <- matrix(sample(1:3,26*3,replace=TRUE),ncol=26)
colnames(M) <- letters
rownames(M) <- c("Huey", "Dewie", "Louie")
subvec(kahle(r=3,p=1:3),M)  # supply a matrix

varchange(as.mvp("1+x+xy + x*y"),x="newx") # variable xy unchanged

kahle(5,3,1:3) |> subs(a="a + delta")

varchange(p,a="]")  # nonstandard variable names OK

varchange_formal(p,"\\]","a")
```

---

summary *Summary methods for mvp objects*

Description

Summary methods for mvp objects and extraction of typical terms

Usage

```
## S3 method for class 'mvp'
summary(object, ...)
## S3 method for class 'summary.mvp'
print(x, ...)
rtypical(object,n=3)
```

Arguments

x, object	Multivariate polynomial, class mvp
n	In rtypical(), number of terms (in addition to the constant) to select
...	Further arguments, currently ignored

Details

The summary method prints out a list of interesting facts about an mvp object such as the longest term or highest power. Function rtypical() extracts the constant if present, and a *random* selection of terms of its argument.

Author(s)

Robin K. S. Hankin

Examples

```
summary(rmvp(40))
rtypical(rmvp(40))
```

---

zero *The zero polynomial*

Description

Test for a multivariate polynomial being zero

Usage

```
is.zero(x)
```

Arguments

x	Object of class mvp

Details

Function is.zero() returns TRUE if x is indeed the zero polynomial. It is defined as length(vars(x))==0 for reasons of efficiency, but conceptually it returns x==constant(0).

(Use constant(0) to create the zero polynomial).

Note

I would have expected the zero polynomial to be problematic (cf the **freegroup** and **permutations** packages, where similar issues require extensive special case treatment). But it seems to work fine, which is a testament to the robust coding in the STL.

A general `mvp` object is something like

`{{"x" -> 3, "y" -> 5} -> 6, {"x" -> 1, "z" -> 8} -> -7}`

which would be $6x^3y^5 - 7xz^8$. The zero polynomial is just `{}`. Neat, eh?

Author(s)

Robin K. S. Hankin

See Also

`constant`

Examples

```
constant(0)

t1 <- as.mvp("x+y")
t2 <- as.mvp("x-y")

stopifnot(is.zero(t1*t2-as.mvp("x^2-y^2")))
```

Index

∗ **datasets**
letters, 14
∗ **math**
summary, 27
∗ **package**
mvp-package, 2
∗ **symbolmath**
allvars, 3
coeffs, 4
deriv, 8
horner, 10
kahle, 12
knight, 13
lowlevel, 15
mpoly, 16
Ops.mvp, 19
print, 20
series, 22
special, 24
subs, 25
zero, 28
%~% *(coeffs)*, 4

a *(letters)*, 14
accessors *(coeffs)*, 4
aderiv *(deriv)*, 8
aderiv_mvp *(deriv)*, 8
allvars, 3
as.function.mvp, 4
as.mpoly.mvp *(mpoly)*, 16
as.mvp *(mvp)*, 16
as_coeffs *(coeffs)*, 4

b *(letters)*, 14

c *(letters)*, 14
coefficients *(coeffs)*, 4
coeffs, 4
coeffs<- *(coeffs)*, 4
consistent *(coeffs)*, 4
constant, 7, *25*, *29*
constant<- *(constant)*, 7

d *(letters)*, 14
deriv, 8, *23*
deriv_mvp *(deriv)*, 8
drop, 9, *27*
drop,mvp-method *(drop)*, 9
drop_mvp *(drop)*, 9

e *(letters)*, 14

f *(letters)*, 14

g *(letters)*, 14

h *(letters)*, 14
hash *(coeffs)*, 4
homog *(special)*, 24
horner, 10, *18*

i *(letters)*, 14
invert, 11
is.coeffs *(coeffs)*, 4
is.constant *(constant)*, 7
is.mvp *(mvp)*, 16
is.zero *(zero)*, 28
is_ok_mvp *(mvp)*, 16

j *(letters)*, 14

k *(letters)*, 14
kahle, 12
knight, 13
knight_mvp *(knight)*, 13

l *(letters)*, 14
letters, 14
lettersymbols *(letters)*, 14
linear *(special)*, 24
lose *(drop)*, 9

lowlevel, 15, *20*

m *(*letters*)*, 14
mpoly, 16
mpoly_to_mvp *(*mpoly*)*, 16
mvp, 16
mvp-class *(*mvp*)*, 16
mvp-package, 2
mvp_add *(*lowlevel*)*, 15
mvp_deriv *(*lowlevel*)*, 15
mvp_eq_mvp *(*Ops.mvp*)*, 19
mvp_modulo *(*Ops.mvp*)*, 19
mvp_negative *(*Ops.mvp*)*, 19
mvp_plus_mvp *(*Ops.mvp*)*, 19
mvp_plus_numeric *(*Ops.mvp*)*, 19
mvp_plus_scalar *(*Ops.mvp*)*, 19
mvp_power *(*lowlevel*)*, 15
mvp_power_scalar *(*Ops.mvp*)*, 19
mvp_prod *(*lowlevel*)*, 15
mvp_subs_mvp *(*subs*)*, 25
mvp_substitute *(*lowlevel*)*, 15
mvp_substitute_mvp *(*lowlevel*)*, 15
mvp_taylor_allvars *(*series*)*, 22
mvp_taylor_onepower_onevar *(*series*)*, 22
mvp_taylor_onevar *(*series*)*, 22
mvp_times_mvp *(*Ops.mvp*)*, 19
mvp_times_scalar *(*Ops.mvp*)*, 19
mvp_to_mpoly *(*mpoly*)*, 16
mvp_to_series *(*series*)*, 22
mvp_vectorised_substitute *(*lowlevel*)*, 15
mvp_vectorized_substitute *(*lowlevel*)*, 15

n *(*letters*)*, 14
namechanger *(*subs*)*, 25
nterms *(*summary*)*, 27
numeric_to_mvp *(*special*)*, 24

o *(*letters*)*, 14
onevarpow *(*series*)*, 22
ooom, *10*, 18
Ops *(*Ops.mvp*)*, 19
Ops.coeffs *(*coeffs*)*, 4
Ops.mvp, 19
Ops.mvp_coeffs *(*coeffs*)*, 4

p *(*letters*)*, 14
powers *(*coeffs*)*, 4
print, 20
print.coeffs *(*coeffs*)*, 4
print.mvp_coeffs *(*coeffs*)*, 4
print.series *(*series*)*, 22
print.summary.mvp *(*summary*)*, 27
print_mvp *(*print*)*, 20
product *(*special*)*, 24

q *(*letters*)*, 14

r *(*letters*)*, 14
rhmvp *(*rmvp*)*, 21
rmvp, 21
rmvpp *(*rmvp*)*, 21
rmvppp *(*rmvp*)*, 21
rtypical *(*summary*)*, 27

s *(*letters*)*, 14
series, 22
simplify *(*lowlevel*)*, 15
special, *12*, 24
subs, *9*, *11*, 25
subs_mvp *(*subs*)*, 25
subsmvp *(*subs*)*, 25
substitute *(*subs*)*, 25
subsy *(*subs*)*, 25
subvec *(*subs*)*, 25
summary, 27

t *(*letters*)*, 14
taylor, *8*
taylor *(*series*)*, 22
trunc *(*series*)*, 22
trunc1 *(*series*)*, 22
truncall *(*series*)*, 22

u *(*letters*)*, 14

v *(*letters*)*, 14
varchange *(*subs*)*, 25
varchange_formal *(*subs*)*, 25
vars *(*coeffs*)*, 4

w *(*letters*)*, 14

x *(*letters*)*, 14
xyz *(*special*)*, 24

y *(*letters*)*, 14

z *(*letters*)*, 14
zero, *25*, 28