Package 'Deriv'

June 11, 2018

Type Package

Title Symbolic Differentiation

Version 3.8.5

Date 2018-06-11

Description R-based solution for symbolic differentiation. It admits user-defined function as well as function substitution in arguments of functions to be differentiated. Some symbolic simplification is part of the work.

License GPL (>= 3)

Suggests testthat

BugReports https://github.com/sgsokol/Deriv/issues

RoxygenNote 5.0.1

NeedsCompilation no

Author Andrew Clausen [aut], Serguei Sokol [aut, cre]

Maintainer Serguei Sokol <sokol@insa-toulouse.fr>

Repository CRAN

Date/Publication 2018-06-11 20:33:29 UTC

R topics documented:

`Deriv-package` *Symbolic Differentiation*

Description

R already contains two differentiation functions: D and deriv.

R's existing functions have several limitations:

- the derivatives table can't be modified at runtime, and is only available in C.
- function cannot substitute function calls. eg:
  `f <- function(x, y) x + y; deriv(~f(x, x^2), "x")`

The advantages of this package include:

- It is entirely written in R, so would be easier to maintain.
- Can differentiate function calls:
  - if the function is in the derivative table, then the chain rule is applied.
  - if the function is not in the derivative table (or it is anonymous), then the function body is substituted in.
  - these two methods can be mixed. An entry in the derivative table need not be self-contained – you don't need to provide an infinite chain of derivatives.
- It's easy to add custom entries to the derivatives table, e.g.
  `drule[["cos"]] <- alist(x=-sin(x))`
- The output can be an executable function, which makes it suitable for use in optimization problems.

Details

Package:	Deriv
Type:	Package
Version:	3.8.5
Date:	2018-06-11
License:	GPL (>= 3)

Two main functions are Deriv() for differentiating and Simplify() for simplifying symbolically.

Author(s)

Andrew Clausen, Serguei Sokol

Maintainer: Serguei Sokol (sokol at insa-toulouse.fr)

References

https://andrewclausen.net/computing/deriv.html

See Also

D, deriv, packages Ryacas, rSymPy

Examples

```
## Not run: f <- function(x) x^2
## Not run: Deriv(f)
# function (x)
# 2 * x
```

`Deriv` *Symbollic differentiation of an expression or function*

Description

Symbollic differentiation of an expression or function

Usage

```
Deriv(f, x = if (is.function(f)) NULL else all.vars(if (is.character(f))
  parse(text = f) else f), env = if (is.function(f)) environment(f) else
  parent.frame(), use.D = FALSE, cache.exp = TRUE, nderiv = NULL,
  combine = "c")
```

Arguments

`f`	An expression or function to be differentiated. f can be • a user defined function: `function(x) x**n` • a string: `"x**n"` • an expression: `expression(x**n)` • a call: `call("^", quote(x), quote(n))` • a language: `quote(x**n)` • a right hand side of a formula: `~ x**n` or `y ~ x**n`
`x`	An optional character vector with variable name(s) with resptect to which `f` must be differentiated. If not provided (i.e. x=NULL), x is guessed either from\ codenames(formals(f)) (if `f` is a function) or from all variables in `f` in other cases. To differentiate expressions including components of lists or vectors, i.e. by expressions like `p[1]`, `theta[["alpha"]]` or `theta$beta`, the vector of variables `x` must be a named vector. For the cited examples, `x` must be given as follows `c(p="1", theta="alpha", theta="beta")`. Note the repeated name `theta` which must be provided for every component of the list `theta` by which a differerentiation is required.

env	An environment where the symbols and functions are searched for. Defaults to parent.frame() for f expression and to environment(f) if f is a function. For primitive function, it is set by default to .GlobalEnv
use.D	An optional logical (default FALSE), indicates if base::D() must be used for differentiation of basic expressions.
cache.exp	An optional logical (default TRUE), indicates if final expression must be optimized with cached subexpressions. If enabled, repeated calculations are made only once and their results stored in cache variables which are then reused.
nderiv	An optional integer vector of derivative orders to calculate. Default NULL value correspond to one differentiation. If length(nderiv)>1, the resulting expression is a list where each component corresponds to derivative order given in nderiv. Value 0 corresponds to the original function or expression non differentiated. All values must be non negative. If the entries in nderiv are named, their names are used as names in the returned list. Otherwise the value of nderiv component is used as a name in the resulting list.
combine	An optional character scalar, it names a function to combine partial derivatives. Default value is "c" but other functions can be used, e.g. "cbind" (cf. Details, NB3), "list" or user defined ones. It must accept any number of arguments or at least the same number of arguments as there are items in x.

Details

R already contains two differentiation functions: D and deriv. D does simple univariate differentiation. "deriv" uses D to do multivariate differentiation. The output of "D" is an expression, whereas the output of "deriv" can be an executable function.

R's existing functions have several limitations. They can probably be fixed, but since they are written in C, this would probably require a lot of work. Limitations include:

- The derivatives table can't be modified at runtime, and is only available in C.
- Function cannot substitute function calls. eg: f <- function(x, y) x + y; deriv(~f(x, x^2), "x")

So, here are the advantages of this implementation:

- It is entirely written in R, so would be easier to maintain.
- Can do multi-variate differentiation.
- Can differentiate function calls:
  - if the function is in the derivative table, then the chain rule is applied. For example, if you declared that the derivative of sin is cos, then it would figure out how to call cos correctly.
  - if the function is not in the derivative table (or it is anonymous), then the function body is substituted in.
  - these two methods can be mixed. An entry in the derivative table need not be self-contained – you don't need to provide an infinite chain of derivatives.
- It's easy to add custom entries to the derivatives table, e.g.

  ```
  drule[["cos"]] <- alist(x=-sin(x))
  ```

  The chain rule will be automatically applied if needed.
- The output is an executable function, which makes it suitable for use in optimization problems.

- Compound functions (i.e. piece-wise functions based on if-else operator) can be differentiated (cf. examples section).
- in case of multiple derivatives (e.g. gradient and hessian calculation), caching can make calculation economies for both

Two work environments `drule` and `simplifications` are exported in the package namescape. As their names indicate, they contain tables of derivative and simplification rules. To see the list of defined rules do `ls(drule)`. To add your own derivative rule for a function called say `sinpi(x)` calculating sin(pi*x), do `drule[["sinpi"]] <- alist(x=pi*cospi(x))`. Here, "x" stands for the first and unique argument in `sinpi()` definition. For a function that might have more than one argument, e.g. `log(x, base=exp(1))`, the drule entry must be a list with a named rule per argument. See `drule$log` for an example to follow. After adding `sinpi` you can differentiate expressions like `Deriv(~ sinpi(x^2), "x")`. The chain rule will automatically apply.

NB. In `abs()` and `sign()` function, singularity treatment at point 0 is left to user's care. For example, if you need NA at singular points, you can define the following: `drule[["abs"]] <- alist(x=ifelse(x==0, NA, sig`
`drule[["sign"]] <- alist(x=ifelse(x==0, NA, 0))`

NB2. In Bessel functions, derivatives are calculated only by the first argument, not by the `nu` argument which is supposed to be constant.

NB3. There is a side effect with vector length. E.g. in `Deriv(~a+b*x, c("a", "b"))` the result is `c(a = 1, b = x)`. To avoid the difference in lengths of a and b components (when x is a vector), one can use an optional parameter `combine` `Deriv(~a+b*x, c("a", "b"), combine="cbind")` which gives `cbind(a = 1, b = x)` producing a two column matrix which is probably the desired result here.
Another example illustrating a side effect is a plain linear regression case and its Hessian: `Deriv(~sum((a+b*x - y)**2), c` producing just a constant `2` for double differentiation by `a` instead of expected result `2*length(x)`. It comes from a simplification of an expression `sum(2)` where the constant is not repeated as many times as length(x) would require it. Here, using the same trick with `combine="cbind"` would not help as all 4 derivatives are just scalars. Instead, one should modify the previous call to explicitly use a constant vector of appropriate length: `Deriv(~sum((rep(a, length(x))+b*x - y)**2), c("a", "b"), n=2)`

Value

- a function if `f` is a function
- an expression if `f` is an expression
- a character string if `f` is a character string
- a language (usually a so called 'call' but may be also a symbol or just a numeric) for other types of `f`

Author(s)

Andrew Clausen (original version) and Serguei Sokol (actual version and maintainer)

Examples

```
## Not run: f <- function(x) x^2
## Not run: Deriv(f)
```

```
# function (x)
# 2 * x

## Not run: f <- function(x, y) sin(x) * cos(y)
## Not run: Deriv(f)
# function (x, y)
# c(x = cos(x) * cos(y), y = -(sin(x) * sin(y)))

## Not run: f_ <- Deriv(f)
## Not run: f_(3, 4)
#              x         y
# [1,] 0.6471023 0.1068000

## Not run: Deriv(~ f(x, y^2), "y")
# -(2 * (y * sin(x) * sin(y^2)))

## Not run: Deriv(quote(f(x, y^2)), c("x", "y"), cache.exp=FALSE)
# c(x = cos(x) * cos(y^2), y = -(2 * (y * sin(x) * sin(y^2))))

## Not run: Deriv(expression(sin(x^2) * y), "x")
# expression(2*(x*y*cos(x^2)))

Deriv("sin(x^2) * y", "x") # differentiate only by x
"2 * (x * y * cos(x^2))"

Deriv("sin(x^2) * y", cache.exp=FALSE) # differentiate by all variables (here by x and y)
"c(x = 2 * (x * y * cos(x^2)), y = sin(x^2))"

# Compound function example (here abs(x) smoothed near 0)
fc <- function(x, h=0.1) if (abs(x) < h) 0.5*h*(x/h)**2 else abs(x)-0.5*h
Deriv("fc(x)", "x", cache.exp=FALSE)
"if (abs(x) < h) x/h else sign(x)"

# Example of a first argument that cannot be evaluated in the current environment:
## Not run:
  suppressWarnings(rm("xx", "yy"))
  Deriv(xx^2+yy^2)

## End(Not run)
# c(xx = 2 * xx, yy = 2 * yy)

# Automatic differentiation (AD), note itermediate variable 'd' assignment
## Not run: Deriv(~{d <- ((x-m)/s)^2; exp(-0.5*d)}, "x")
#{
#   d <- ((x - m)/s)^2
#   .d_x <- 2 * ((x - m)/s^2)
#   -(0.5 * (.d_x * exp(-(0.5 * d))))
#}

# Custom derivation rule
## Not run:
  myfun <- function(x, y=TRUE) NULL # do something usefull
  dmyfun <- function(x, y=TRUE) NULL # myfun derivative by x.
```

```
  drule[["myfun"]] <- alist(x=dmyfun(x, y), y=NULL) # y is just a logical
  Deriv(myfun(z^2, FALSE), "z")
  # 2 * (z * dmyfun(z^2, FALSE))

## End(Not run)
# Differentiantion by list components
## Not run:
  theta <- list(m=0.1, sd=2.)
  x <- names(theta)
  names(x)=rep("theta", length(theta))
  Deriv(~exp(-(x-theta$m)**2/(2*theta$sd)), x, cache.exp=FALSE)
# c(theta_m = exp(-((x - theta$m)^2/(2 * theta$sd))) *
#  (x - theta$m)/theta$sd, theta_sd = 2 * (exp(-((x - theta$m)^2/
#  (2 * theta$sd))) * (x - theta$m)^2/(2 * theta$sd)^2))

## End(Not run)
```

`format1` *Wrapper for base::format() function*

Description

Wrapper for base::format() function

Usage

```
format1(expr)
```

Arguments

`expr`	An expression or symbol or language to be converted to a string.

Value

A character vector of length 1 contrary to base::format() which can split its output over several lines.

`Simplify` *Symbollic simplification of an expression or function*

Description

Symbollic simplification of an expression or function

Usage

```
Simplify(expr, env = parent.frame(), scache = new.env())

Cache(st, env = Leaves(st), prefix = "")

deCache(st)
```

Arguments

expr	An expression to be simplified, expr can be • an expression: `expression(x+x)` • a string: `"x+x"` • a function: `function(x) x+x` • a right hand side of a formula: `~x+x` • a language: `quote(x+x)`
env	An environment in which a simplified function is created if `expr` is a function. This argument is ignored in all other cases.
scache	An environment where there is a list in which simplified expression are cached
st	A language expression to be cached
prefix	A string to start the names of the cache variables

Details

An environment `simplifications` containing simplification rules, is exported in the namespace accessible by the user. Cache() is used to remove redundunt calculations by storing them in cache variables. Default parameters to Cache() does not have to be provided by user. deCache() makes the inverse job – a series of assignements are replaced by only one big expression without assignement. Sometimes it is usefull to apply deChache() and only then pass its result to Cache().

Value

A simplified expression. The result is of the same type as `expr` except for formula, where a language is returned.

Index