

Package ‘Deriv’

September 20, 2019

Type Package

Title Symbolic Differentiation

Version 3.9.0

Date 2019-09-20

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] (<<https://orcid.org/0000-0002-5674-3327>>)

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

Repository CRAN

Date/Publication 2019-09-20 16:40:18 UTC

R topics documented:

Deriv-package	2
Deriv	3
format1	7
Simplify	7
Index	9

Description

R already contains two differentiation functions: `D` and `deriv`.

These 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: 3.9.0
Date: 2019-09-20
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 | <p>An expression or function to be differentiated. f can be</p> <ul style="list-style-type: none"> • a user defined function: <code>function(x) x**n</code> • a string: <code>"x**n"</code> • an expression: <code>expression(x**n)</code> • a call: <code>call("^",quote(x),quote(n))</code> • a language: <code>quote(x**n)</code> • a right hand side of a formula: <code>~ x**n</code> or <code>y ~ x**n</code> |
| x | <p>An optional character vector with variable name(s) with respect to which f must be differentiated. If not provided (i.e. <code>x=NULL</code>), x is guessed either from <code>codenames(formals(f))</code> (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 <code>p[1]</code>, <code>theta[["alpha"]]</code> or <code>theta\$beta</code>, the vector of variables x must be a named vector. For the cited examples, x must be given as follows <code>c(p="1", theta="alpha", theta="beta")</code>. Note the repeated name theta which must be provided for every component of the list theta by which a differentiation is required.</p> |

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

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 namespace. 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,sign(x))`
`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("a","b"),n=c(hessian=2))` 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 intermediate 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)
#   -(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)
# Differentiation 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

<code>expr</code>	An expression to be simplified, <code>expr</code> can be <ul style="list-style-type: none">• an expression: <code>expression(x+x)</code>• a string: <code>"x+x"</code>• a function: <code>function(x) x+x</code>• a right hand side of a formula: <code>~x+x</code>• a language: <code>quote(x+x)</code>
<code>env</code>	An environment in which a simplified function is created if <code>expr</code> is a function. This argument is ignored in all other cases.
<code>scache</code>	An environment where there is a list in which simplified expression are cached
<code>st</code>	A language expression to be cached
<code>prefix</code>	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 redundant 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 assignments are replaced by only one big expression without assignment. Sometimes it is useful to apply `deCache()` 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

*Topic **package**

Deriv-package, 2

Cache (Simplify), 7

D, 3

deCache (Simplify), 7

Deriv, 3

deriv, 3

Deriv-package, 2

drule (Deriv), 3

format1, 7

simplifications (Simplify), 7

Simplify, 7