Package ‘Deriv’

December 10, 2019

Type Package
Title Symbolic Differentiation
Version 4.0
Date 2019-12-10
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 6.1.1
Imports methods
NeedsCompilation no
Author Andrew Clausen [aut],
Serguei Sokol [aut, cre] (<https://orcid.org/0000-0002-5674-3327>),
Andreas Rappold [ctb]
Maintainer Serguei Sokol <sokol@insa-toulouse.fr>
Repository CRAN
Date/Publication 2019-12-10 15:50:09 UTC

R topics documented:

Deriv-package .................................................. 2
Deriv .............................................................. 3
format1 ............................................................ 7
Simplify ............................................................ 8

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:
  \[ f <- \text{function}(x,y) \ x + y; \ \text{deriv}(-f(x,x^2),"x") \]

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.
  \[ \text{drule["cos"] <- alist(x = -sin(x))} \]
- The output can be an executable function, which makes it suitable for use in optimization problems.
- Starting from v4.0, some matrix calculus operations are possible (contribution of Andreas Rappold). See an example in help("Deriv") for differentiation of the inverse of 2x2 matrix and whose elements depend on variable of differentiation \( x \).

Details

| Package: | Deriv |
| Type: | Package |
| Version: | 4.0 |
| Date: | 2019-12-10 |
| License: | GPL (>= 3) |

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

Author(s)

Andrew Clausen, Serguei Sokol
Deriv

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

References

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

See Also

D, deriv, packages Ryacas, rSymPy

Examples

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

---

Symbolic differentiation of an expression or function

Description

Symbolic differentiation of an expression or function

Usage

```r
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 respect 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 differentiation 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 sub-expressions. 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
• Starting from v4.0, some matrix calculus operations are possible (contribution of Andreas Rappold). See an example hereafter for differentiation of the inverse of 2x2 matrix and whose elements depend on variable of differentiation x.

Two environments drule and simplifications are exported in the package’s 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*cospix)). 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. Starting from v4.0, you can differentiate expressions like Deriv(~ sinpi(x^2),"x"). The chain rule will automatically apply.

Starting from v4.0, user can benefit from a syntax .d_X in the rule writing. Here X must be replace by an argument name (cf. drule["solve"] for an example). A use of this syntax leads to a replacement of this place-holder by a derivative of the function (chain rule is automatically integrated) by the named argument.

Another v4.0 novelty in rule’s syntax is a possible use of optional parameter "/grave.Var_missing" which can be set to TRUE or FALSE (default) to indicate how to treat missing arguments. By default, i.e. in absence of this parameter or set to FALSE, missing arguments were replaced by their default values. Now, if `/grave.Var_missing`=TRUE is specified in a rule, the missing arguments will be left missed in the derivative. Look drule["solve"] for an example.

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))

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

```r
## 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)
# [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", cache.exp=FALSE)
##
## 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 useful
dmyfun <- function(x, y=TRUE) NULL # myfun derivative by x.
drule[["myfun"]]<-alist(x=dmyfun(x, 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)
## Differentiation in matrix calculus
## Not run:
Deriv(~solve(matrix(c(1, x, x**2, x**3), nrow=2, ncol=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.
Simplify

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

Simplify

Symbolic simplification of an expression or function

Description

Symbolic simplification of an expression or function

Usage

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

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

deCache(st)

Arguments

eexpr

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 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), 8

D, 3
decache (Simplify), 8
Deriv, 3
deriv, 3
Deriv-package, 2
drule (Deriv), 3

format1, 7

simplifications (Simplify), 8
Simplify, 8