nlsr Derivatives

John C. Nash

2019-08-29

This vignette is an attempt to catalog and illustrate the various capabilities in the **R** statistical computing system to perform differentiation. There are many traps and pitfalls for the unwary in doing this, and it is hoped that this rather long treatment will serve to record these and show how to avoid them, and how to reliably compute the derivatives desired. Derivative capabilities of **R** are in the base system (essentially the functions `D()` and `deriv()`) and in different packages, namely `nlsr`, `Deriv`, `Ryacas`. General tools for approximations to derivatives are found in the package `numDeriv` as well as `optextras`. Particular approximations may be embedded in various packages, but not necessarily exported for use in scripts or packages.

As a way of recording where attention is needed either to this document or to the functions and methods described, I have put double question marks in various places.

Note: To distinguish output results (which are prefaced '`##`' by **knitr**, comments in the **R** code are prefaced by '`#-`'.)

Available analytic differentiation tools

R has a number of tools for finding analytic derivatives.

- **stats**: tools `D()` and `deriv()` (R Development Core Team (2008))
- **nlsr**: tools `nlsDeriv()`, `fnDeriv()`, and possibly ?? `model2rjfun` (Nash and Murdoch (2019))
- **Deriv**: tools `Deriv()` (Clausen and Sokol (2018))
- **Ryacas**: tools ?? Goedman et al. (2019)
- In 2018, Changcheng Li conducted a Google Summer of Code project to link R to Julia's Automatic Differentiation tools, resulting in the experimental package `autodiffr` (see https://github.com/Non-Contradiction/autodiffr).
- ?? any other packages that give analytic derivatives?

How the tools are used

This is an overview section to give an idea of the capabilities. It is not intended to be exhaustive, but to give pointers to how the tools can be used quickly.

An important issue that may cause a lot of difficulty is the iterating of the tools. That is, we compute a derivative, then want to apply a tool to the derivative to get a second derivative. In doing so, we need to be careful that the type (class??) of the quantity output by the tool is passed back into the tool in a form that will generate a derivative expression. Some examples are presented.

We also note that the **Deriv** package will give a result in cases when the input is undefined. This is clearly a bug.

stats

`D()`, `deriv()` and `deriv3()`: As `deriv3()` is stated to be the same as `deriv()` but with argument `hessian=TRUE`, we will for now only consider the first two.

```r
dx2x <- deriv(~ x^2, "x")
dx2x
```

```
## expression({
##     .value <- x^2
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 2 * x
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```r
mode(dx2x)
```

```
## [1] "expression"
```

```r
str(dx2x)
```

```
##   expression({  .value <- x^2  .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))  .grad[,
```

```r
x <- -1:2
eval(dx2x) # This is evaluated at -1, 0, 1, 2, with the result in the "gradient" attribute
```

```
## [1] 1 0 1 4
## attr(,"gradient")
##       x
## [1,] -2
## [2,]  0
## [3,]  2
## [4,]  4
```

```r
# Note that we cannot (easily) differentiate this again.
firstd <- attr(dx2x,"gradient")
str
```

```
## function (object, ...)
## UseMethod("str")
## <bytecode: 0x556106cfadc0>
## <environment: namespace:utils>
```

```r
#  ... and the following gives an error
d2x2x <- try(deriv(firstd, "x"))
```

```
## Error in deriv.default(firstd, "x") :
##   invalid expression in 'FindSubexprs'
```

```r
str(d2x2x)
```

```
##  'try-error' chr "Error in deriv.default(firstd, \"x\") : \n  invalid expression in 'FindSubexprs'\n
##  - attr(*, "condition")=List of 2
##   ..$ message: chr "invalid expression in 'FindSubexprs'"
##   ..$ call   : language deriv.default(firstd, "x")
##   ..- attr(*, "class")= chr [1:3] "simpleError" "error" "condition"
```

```r
#- Build a function from the expression
fdx2x<-function(x){eval(dx2x)}
fdx2x(1)
```

```
## [1] 1
## attr(,"gradient")
##      x
```

```
## [1,] 2
```

```
fdx2x(3.21)
```

```
## [1] 10.3041
## attr(,"gradient")
##         x
## [1,] 6.42
```

```
fdx2x(1:5)
```

```
## [1]  1  4  9 16 25
## attr(,"gradient")
##       x
## [1,]  2
## [2,]  4
## [3,]  6
## [4,]  8
## [5,] 10
```

```
#- # Now try D()
Dx2x <- D(expression(x^2), "x")
Dx2x
```

```
## 2 * x
```

```
x <- -1:2
eval(Dx2x)
```

```
## [1] -2  0  2  4
```

```
# We can differentiate aggain
D2x2x <- D(Dx2x,"x")
D2x2x
```

```
## [1] 2
```

```
eval(D2x2x) #- But we don't get a vector -- could be an issue in gradients/Jacobians
```

```
## [1] 2
```

```
#- Note how we handle an expression stored in a string via parse(text=  ))
sx2 <- "x^2"
sDx2x <- D(parse(text=sx2), "x")
sDx2x
```

```
## 2 * x
```

```
#- But watch out! The following "seems" to work, but the answer is not as intended.  The problem is tha
# argument is evaluated before being used.  Since
# x exists, it fails
x
```

```
## [1] -1  0  1  2
```

```
Dx2xx <- D(x^2, "x")
Dx2xx
```

```
## [1] 0
```

```
eval(Dx2xx)
```

```
## [1] 0
```

```r
#-  Something 'tougher':
trig.exp <- expression(sin(cos(x + y^2)))
( D.sc <- D(trig.exp, "x") )
```

```
## -(cos(cos(x + y^2)) * sin(x + y^2))
```

```r
all.equal(D(trig.exp[[1]], "x"), D.sc)
```

```
## [1] TRUE
```

```r
( dxy <- deriv(trig.exp, c("x", "y")) )
```

```
## expression({
##     .expr2 <- x + y^2
##     .expr3 <- cos(.expr2)
##     .expr5 <- cos(.expr3)
##     .expr6 <- sin(.expr2)
##     .value <- sin(.expr3)
##     .grad <- array(0, c(length(.value), 2L), list(NULL, c("x",
##         "y")))
##     .grad[, "x"] <- -(.expr5 * .expr6)
##     .grad[, "y"] <- -(.expr5 * (.expr6 * (2 * y)))
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```r
y <- 1
eval(dxy)
```

```
## [1]  0.8414710  0.5143953 -0.4042392 -0.8360219
## attr(,"gradient")
##               x         y
## [1,]  0.0000000  0.000000
## [2,] -0.7216061 -1.443212
## [3,] -0.8316919 -1.663384
## [4,] -0.0774320 -0.154864
```

```r
eval(D.sc)
```

```
## [1]  0.0000000 -0.7216061 -0.8316919 -0.0774320
```

```r
#-  function returned:
deriv((y ~ sin(cos(x) * y)), c("x","y"), func = TRUE)
```

```
## function (x, y)
## {
##     .expr1 <- cos(x)
##     .expr2 <- .expr1 * y
##     .expr4 <- cos(.expr2)
##     .value <- sin(.expr2)
##     .grad <- array(0, c(length(.value), 2L), list(NULL, c("x",
##         "y")))
##     .grad[, "x"] <- -(.expr4 * (sin(x) * y))
##     .grad[, "y"] <- .expr4 * .expr1
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
#- ??#-  Surely there is an error, since documentation says no lhs! i.e.,
#- "expr: a 'expression' or 'call' or (except 'D') a formula with no lhs."
#-  function with defaulted arguments:
(fx <- deriv(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
             function(b0, b1, th, x = 1:7){} ) )
```

```
## function (b0, b1, th, x = 1:7)
## {
##     .expr3 <- 2^(-x/th)
##     .value <- b0 + b1 * .expr3
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("b0",
##         "b1", "th")))
##     .grad[, "b0"] <- 1
##     .grad[, "b1"] <- .expr3
##     .grad[, "th"] <- b1 * (.expr3 * (log(2) * (x/th^2)))
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
fx(2, 3, 4)
```

```
## [1] 4.522689 4.121320 3.783811 3.500000 3.261345 3.060660 2.891905
## attr(,"gradient")
##      b0        b1        th
## [1,]  1 0.8408964 0.1092872
## [2,]  1 0.7071068 0.1837984
## [3,]  1 0.5946036 0.2318331
## [4,]  1 0.5000000 0.2599302
## [5,]  1 0.4204482 0.2732180
## [6,]  1 0.3535534 0.2756976
## [7,]  1 0.2973018 0.2704720
```

```
#-  First derivative
D(expression(x^2), "x")
```

```
## 2 * x
```

```
#-  stopifnot(D(as.name("x"), "x") == 1) #- A way of testing.
#- This works by coercing "x" to name/symbol, and derivative should be 1.
#- Would fail only if "x" cannot be so coerced. How could this happen??
#-  Higher derivatives showing deriv3
myd3 <- deriv3(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
     c("b0", "b1", "th", "x") )
myd3(2,3,4, x=1:7)
```

```
## [1] 4.522689 4.121320 3.783811 3.500000 3.261345 3.060660 2.891905
## attr(,"gradient")
##      b0        b1        th
## [1,]  1 0.8408964 0.1092872
## [2,]  1 0.7071068 0.1837984
## [3,]  1 0.5946036 0.2318331
## [4,]  1 0.5000000 0.2599302
## [5,]  1 0.4204482 0.2732180
## [6,]  1 0.3535534 0.2756976
## [7,]  1 0.2973018 0.2704720
## attr(,"hessian")
```

```
## , , b0
##
##      b0 b1 th
## [1,]  0  0  0
## [2,]  0  0  0
## [3,]  0  0  0
## [4,]  0  0  0
## [5,]  0  0  0
## [6,]  0  0  0
## [7,]  0  0  0
##
## , , b1
##
##      b0 b1         th
## [1,]  0  0 0.03642906
## [2,]  0  0 0.06126613
## [3,]  0  0 0.07727771
## [4,]  0  0 0.08664340
## [5,]  0  0 0.09107265
## [6,]  0  0 0.09189920
## [7,]  0  0 0.09015733
##
## , , th
##
##      b0         b1          th
## [1,]  0 0.03642906 -0.04990909
## [2,]  0 0.06126613 -0.07597428
## [3,]  0 0.07727771 -0.08578635
## [4,]  0 0.08664340 -0.08492263
## [5,]  0 0.09107265 -0.07742765
## [6,]  0 0.09189920 -0.06618667
## [7,]  0 0.09015733 -0.05321485
```

```
#- check against deriv()
myd3a <- deriv(y ~ b0 + b1 * 2^(-x/th), c("b0", "b1", "th"),
     c("b0", "b1", "th", "x"), hessian=TRUE )
myd3a(2,3,4, x=1:7)
```

```
## [1] 4.522689 4.121320 3.783811 3.500000 3.261345 3.060660 2.891905
## attr(,"gradient")
##      b0        b1        th
## [1,]  1 0.8408964 0.1092872
## [2,]  1 0.7071068 0.1837984
## [3,]  1 0.5946036 0.2318331
## [4,]  1 0.5000000 0.2599302
## [5,]  1 0.4204482 0.2732180
## [6,]  1 0.3535534 0.2756976
## [7,]  1 0.2973018 0.2704720
## attr(,"hessian")
## , , b0
##
##      b0 b1 th
## [1,]  0  0  0
## [2,]  0  0  0
## [3,]  0  0  0
```

```
## [4,]  0  0  0
## [5,]  0  0  0
## [6,]  0  0  0
## [7,]  0  0  0
##
## , , b1
##
##      b0 b1         th
## [1,]  0  0 0.03642906
## [2,]  0  0 0.06126613
## [3,]  0  0 0.07727771
## [4,]  0  0 0.08664340
## [5,]  0  0 0.09107265
## [6,]  0  0 0.09189920
## [7,]  0  0 0.09015733
##
## , , th
##
##      b0         b1          th
## [1,]  0 0.03642906 -0.04990909
## [2,]  0 0.06126613 -0.07597428
## [3,]  0 0.07727771 -0.08578635
## [4,]  0 0.08664340 -0.08492263
## [5,]  0 0.09107265 -0.07742765
## [6,]  0 0.09189920 -0.06618667
## [7,]  0 0.09015733 -0.05321485
```

```
identical(myd3a, myd3) #- Remember to check things!
```

```
## [1] TRUE
```

```
#-  Higher derivatives:
DD <- function(expr, name, order = 1) {
   if(order < 1) stop("'order' must be >= 1")
   if(order == 1) D(expr, name)
   else DD(D(expr, name), name, order - 1)
}
DD(expression(sin(x^2)), "x", 3)
```

```
## -(sin(x^2) * (2 * x) * 2 + ((cos(x^2) * (2 * x) * (2 * x) + sin(x^2) *
##     2) * (2 * x) + sin(x^2) * (2 * x) * 2))
```

```
#-  showing the limits of the internal "simplify()" :
#-  -sin(x^2) * (2 * x) * 2 + ((cos(x^2) * (2 * x) * (2 * x) + sin(x^2) *
#-     2) * (2 * x) + sin(x^2) * (2 * x) * 2)
```

nlsr

```
require(nlsr)
```

```
## Loading required package: nlsr
```

```
dx2xn <- nlsDeriv(~ x^2, "x")
dx2xn
```

```
## 2 * x
```

```
mode(dx2xn)
```

```
## [1] "call"
```

```
str(dx2xn)
```

```
##  language 2 * x
```

```
x <- -1:2
eval(dx2xn) # This is evaluated at -1, 0, 1, 2, BUT result is returned directly,
```

```
## [1] -2  0  2  4
```

```
#-  NOT in "gradient" attribute
firstdn <- dx2xn
str(firstdn)
```

```
##  language 2 * x
```

```
d2x2xn <- nlsDeriv(firstdn, "x")
d2x2xn
```

```
## [1] 2
```

```
d2x2xnF <- nlsDeriv(firstdn, "x", do_substitute=FALSE)
d2x2xnF # in this case we get the same result
```

```
## [1] 2
```

```
d2x2xnT <- nlsDeriv(firstdn, "x", do_substitute=TRUE)
d2x2xnT # 0 ## WATCH OUT
```

```
## [1] 0
```

```
#- ?? We can iterate the derivatives
nlsDeriv(d2x2xn, "x")
```

```
## [1] 0
```

```
nlsDeriv(x^2, "x")# 0
```

```
## [1] 0
```

```
nlsDeriv(x^2, "x", do_substitute=FALSE)# 0
```

```
## [1] 0
```

```
nlsDeriv(x^2, "x", do_substitute=TRUE) # 2 * x
```

```
## [1] 0
```

```
nlsDeriv(~ x^2, "x") # 2 * x
```

```
## 2 * x
```

```
nlsDeriv(~ x^2, "x", do_substitute=FALSE) # 2 * x
```

```
## 2 * x
```

```
nlsDeriv(~ x^2, "x", do_substitute=TRUE) # 2 * x
```

```
## [1] 0
```

```
#?? firstde <- quote(firstd)
#?? firstde
```

```
#?? firstde <- bquote(firstd)
#?? firstde
#?? nlsDeriv(firstde, "x")
d2 <- nlsDeriv(2 * x, "x")
str(d2)
```

```
##  num 0
```

```
d2
```

```
## [1] 0
```

```
#?? firstc <- as.call(firstd)
#?? nlsDeriv(firstc, "x")
#- Build a function from the expression
#?? fdx2xn<-function(x){eval(dx2xn)}
#?? fdx2xn(1)
#?? fdx2xn(3.21)
#?? fdx2xn(1:5)
```

The tool `codeDeriv` returns an R expression to evaluate the derivative efficiently. `fnDeriv` wraps it in a function. By default the arguments to the function are constructed from all variables in the expression. In the example below this includes `x`.

```
codeDeriv(parse(text="b0 + b1 * 2^(-x/th)"), c("b0", "b1", "th"))
```

```
## {
##     .expr1 <- -x
##     .expr2 <- .expr1/th
##     .expr3 <- 2^.expr2
##     .value <- b0 + b1 * 2^(.expr2)
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("b0",
##     "b1", "th")))
##     .grad[, "b0"] <- 1
##     .grad[, "b1"] <- .expr3
##     .grad[, "th"] <- b1 * (.expr3 * 0.693147180559945 * -(.expr1/th^2))
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
#- Include parameters as arguments
fj.1 <- fnDeriv(parse(text="b0 + b1 * 2^(-x/th)"), c("b0", "b1", "th"))
head(fj.1)
```

```
##
## 1 function (b0, b1, x, th)
## 2 {
## 3     .expr1 <- -x
## 4     .expr2 <- .expr1/th
## 5     .expr3 <- 2^.expr2
## 6     .value <- b0 + b1 * 2^(.expr2)
```

```
fj.1(1,2,3,4)
```

```
## [1] 2.189207
## attr(,"gradient")
##      b0        b1        th
## [1,]  1 0.5946036 0.1545554
```

```r
#- Get all parameters from the calling environment
fj.2 <- fnDeriv(parse(text="b0 + b1 * 2^(-x/th)"), c("b0", "b1", "th"),
        args = character())
head(fj.2)
```

```
##
## 1 function ()
## 2 {
## 3     .expr1 <- -x
## 4     .expr2 <- .expr1/th
## 5     .expr3 <- 2^.expr2
## 6     .value <- b0 + b1 * 2^(.expr2)
```

```r
b0 <- 1
b1 <- 2
x <- 3
th <- 4
fj.2()
```

```
## [1] 2.189207
## attr(,"gradient")
##      b0        b1        th
## [1,]  1 0.5946036 0.1545554
```

```r
#- Just use an expression
fje <- codeDeriv(parse(text="b0 + b1 * 2^(-x/th)"), c("b0", "b1", "th"))
eval(fje)
```

```
## [1] 2.189207
## attr(,"gradient")
##      b0        b1        th
## [1,]  1 0.5946036 0.1545554
```

```r
dx2xnf <- fnDeriv(~ x^2, "x") #- Use tilde
dx2xnf <- fnDeriv(expression(x^2), "x") #- or use expression()
dx2xnf
```

```
## function (x)
## {
##     .value <- x^2
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 2 * x
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```r
mode(dx2xnf)
```

```
## [1] "function"
```

```r
str(dx2xnf)
```

```
## function (x)
```

```r
x <- -1:2
#?? eval(dx2xnf) # This is evaluated at -1, 0, 1, 2, BUT result is returned directly,
#-  NOT in "gradient" attribute
# Note that we cannot (easily) differentiate this again.
```

```
# firstd <- dx2xnf
# str(firstd)
# d2x2xnf <- try(nlsDeriv(firstd, "x")) #- this APPEARS to work, but WRONG answer
# str(d2x2xnf)
# d2x2xnf
# eval(d2x2xnf)

# dx2xnfh <- fnDeriv(expression(x^2), "x", hessian=TRUE) #- Try for second derivatives
# dx2xnfh
# mode(dx2xnfh)
# str(dx2xnfh)
# x <- -1
# eval(dx2xnfh) # This is evaluated at -1, 0, 1, 2, BUT result is returned directly,
```

Deriv

The following examples are drawn from the `example(Deriv)` contained in the `Deriv` package.

```
require(Deriv)
```

```
## Loading required package: Deriv
```

```
f <- function(x) x^2
Deriv(f)
```

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

```
#- Should see
#- function (x)
#- 2 * x
#- Now save the derivative
f1 <- Deriv(f)
f1 #- print it
```

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

```
f2 <- Deriv(f1) #- and take second derivative
f2 #- print it
```

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

```
f <- function(x, y) sin(x) * cos(y)
f_ <- Deriv(f)
f_ #- print it
```

```
## function (x, y)
## c(x = cos(x) * cos(y), y = -(sin(x) * sin(y)))
```

```
#- Should see
#- function (x, y)
#- c(x = cos(x) * cos(y), y = -(sin(x) * sin(y)))
f_(3, 4)
```

```
##         x         y
```

```
## 0.6471023 0.1068000
```

```
#- Should see
#-                x         y
#- [1,] 0.6471023 0.1068000

f2 <- Deriv(~ f(x, y^2), "y") #- This has a tilde to render the 1st argument as a formula object
#- Also we are substituting in y^2 for y
f2 #- print it
```

```
## -(2 * (y * sin(x) * sin(y^2)))
```

```
#- -(2 * (y * sin(x) * sin(y^2)))
mode(f2) #- check what type of object it is
```

```
## [1] "call"
```

```
arg1 <- ~ f(x,y^2)
mode(arg1) #- check the type
```

```
## [1] "call"
```

```
f2a <- Deriv(arg1, "y")
f2a #- and print to see if same as before
```

```
## -(2 * (y * sin(x) * sin(y^2)))
```

```
#- try evaluation of f using current x and y
x
```

```
## [1] -1  0  1  2
```

```
y
```

```
## [1] 1
```

```
f(x,y^2)
```

```
## [1] -0.4546487  0.0000000  0.4546487  0.4912955
```

```
eval(f2a) #- We need x and y defined to do this.
```

```
## [1]  1.416147  0.000000 -1.416147 -1.530295
```

```
f3 <- Deriv(quote(f(x, y^2)), c("x", "y"), cache.exp=FALSE) #- check cache.exp operation
#- Note that we need to quote or will get evaluation at current x, y values (if they exist)
f3 #- print it
```

```
## c(x = cos(x) * cos(y^2), y = -(2 * (y * sin(x) * sin(y^2))))
```

```
#- c(x = cos(x) * cos(y^2), y = -(2 * (y * sin(x) * sin(y^2))))
f3c <- Deriv(quote(f(x, y^2)), c("x", "y"), cache.exp=TRUE) #- check cache.exp operation
f3c #- print it
```

```
## {
##     .e1 <- y^2
##     c(x = cos(x) * cos(.e1), y = -(2 * (y * sin(x) * sin(.e1))))
## }
```

```
#- Now want to evaluate the results
#- First must provide some data
x <- 3
```

```
y <- 4
eval(f3c)
```

```
##         x         y
## 0.9480757 0.3250313
```

```
#- Should see
#- x         y
#- 0.9480757 0.3250313
eval(f3) #- check this also
```

```
##         x         y
## 0.9480757 0.3250313
```

```
#- or we can create functions
f3cf <- function(x, y){eval(f3c)}
f3cf(x=1, y=2)
```

```
##          x          y
## -0.3531652  2.5473094
```

```
#-          x          y
#- -0.3531652  2.5473094
f3f <- function(x,y){eval(f3)}
f3f(x=3, y=4)
```

```
##         x         y
## 0.9480757 0.3250313
```

```
#-         x         y
#- 0.9480757 0.3250313

#- try an expression
Deriv(expression(sin(x^2) * y), "x")
```

```
## expression(2 * (x * y * cos(x^2)))
```

```
#- should see
#- expression(2 * (x * y * cos(x^2)))

#- quoted string
Deriv("sin(x^2) * y", "x") # differentiate only by x
```

```
## [1] "2 * (x * y * cos(x^2))"
```

```
#- Should see
#- "2 * (x * y * cos(x^2))"

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

```
## [1] "c(x = 2 * (x * y * cos(x^2)), y = sin(x^2))"
```

```
#- Note that default is to differentiate by all variables.
#- Should see
#- "c(x = 2 * (x * y * cos(x^2)), y = sin(x^2))"

#- Compound function example (here abs(x) smoothed near 0)
#- Note that this introduces the possibilty of `if` statements in the code
#- BUT (JN) seems to give back quoted string, so we must parse.
```

```
fc <- function(x, h=0.1) if (abs(x) < h) 0.5*h*(x/h)**2 else abs(x)-0.5*h
efc1 <- Deriv("fc(x)", "x", cache.exp=FALSE)
#- "if (abs(x) < h) x/h else sign(x)"
#- A few checks on the results
efc1
```

```
## [1] "if (abs(x) < h) x/h else sign(x)"
```

```
fc1 <- function(x,h=0.1){ eval(parse(text=efc1)) }
fc1
```

```
## function(x,h=0.1){ eval(parse(text=efc1)) }
```

```
## h=0.1
fc1(1)
```

```
## [1] 1
```

```
fc1(0.001)
```

```
## [1] 0.01
```

```
fc1(-0.001)
```

```
## [1] -0.01
```

```
fc1(-10)
```

```
## [1] -1
```

```
fc1(0.001, 1)
```

```
## [1] 0.001
```

```
#- Example of a first argument that cannot be evaluated in the current environment:
  try(suppressWarnings(rm("xx", "yy"))) #- Make sure there are no objects xx or yy
  Deriv(~ xx^2+yy^2)
```

```
## c(xx = 2 * xx, yy = 2 * yy)
```

```
#- Should show
#- c(xx = 2 * xx, yy = 2 * yy)
#- ?? What is the meaning / purpose of this construct?

#- ?? Is following really AD?
#- Automatic differentiation (AD), note intermediate variable 'd' assignment
Deriv(~{d <- ((x-m)/s)^2; exp(-0.5*d)}, "x")
```

```
## {
##     .e1 <- x - m
##     -(exp(-(0.5 * (.e1/s)^2)) * .e1/s^2)
## }
```

```
# Note that the result we see does NOT match what follows in the example(Deriv) (JN ??)
#{
#   d <- ((x - m)/s)^2
#   .d_x <- 2 * ((x - m)/s^2)
#   -(0.5 * (.d_x * exp(-(0.5 * d))))
#}
#- For some reason the intermediate variable d is NOT included.??
```

```
#- Custom derivative rule. Note that this needs explanations??
  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), y=NULL) #- y is just a logical
  Deriv(myfun(z^2, FALSE), "z")
```

```
## 2 * (z * dmyfun(z^2, FALSE))
```

```
  # 2 * (z * dmyfun(z^2, FALSE))

#- Differentiation by list components
  theta <- list(m=0.1, sd=2.) #- Why do we set values??
  x <- names(theta) #- and why these particular names??
  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))
```

```
#- Should show the following (but why??)
#- 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))
lderiv <-  Deriv(~exp(-(x-theta$m)**2/(2*theta$sd)), x, cache.exp=FALSE)
fld <- function(x){ eval(lderiv)} #- put this in a function
fld(2) #- and evaluate at a value
```

```
##   theta_m  theta_sd
## 0.3852768 0.1830065
```

Deriv has some design choices that can get the user into trouble. The following example shows one such problem.

```
library(Deriv)
rm(x) # ensures x is undefined
Deriv(~ x, "x")  # returns [1] 1 -- clearly a bug!
```

```
## [1] 1
```

```
Deriv(~ x^2, "x")   # returns 2 * x
```

```
## 2 * x
```

```
x <- quote(x^2)
Deriv(x, "x") # returns 2 * x
```

```
## 2 * x
```

By comparison, **nlsr**

```
rm(x) # in case it is defined
library(nlsr)
try(nlsDeriv(x, "x")  ) # fails, not a formula
```

```
## Error : object 'x' not found
```

```
try(nlsDeriv(as.expression("x"), "x")  ) # expression(NULL)
try(nlsDeriv(~x, "x")  ) # 1
```

```
## [1] 1
```

```
try(nlsDeriv(x^2, "x"))   # fails
```

```
## Error : object 'x' not found
```

```
try(nlsDeriv(~x^2, "x")) # 2 * x
```

```
## 2 * x
```

```
x <- quote(x^2)
try(nlsDeriv(x, "x")) # returns 2 * x
```

```
## 2 * x
```

Ryacas

There is at least one other symbolic package for R. Here we look at **Ryacas**. The structures for using yacas tools do not seem at the time of writing (2016-10-21) to be suitable for working with nonlinear least squares or optimization facilities of **R**. Thus, for the moment, we will not pursue the derivatives available in `Ryacas` beyond the following example provided by Gabor Grothendieck.

```
require(nlsr)
dnlsr <- nlsr::nlsDeriv(~ sin(x+y), "x")
print(dnlsr)
```

```
## cos(x + y)
```

```
class(dnlsr)
```

```
## [1] "call"
```

```
detach("package:nlsr", unload=TRUE)
detach("package:Deriv", unload=TRUE)

## New Ryacas mechanism as of 2019-8-29 from mikl@math.aau.dk (Mikkel Meyer Andersen)

yac_str("D(x) Sin(x+y)")
```

```
## [1] "Cos(x+y)"
```

```
# or if an expression is needed:
ex <- yac_expr("D(x) Sin(x+y)")
ex
```

```
## expression(cos(x + y))
```

```
expression(cos(x + y))
```

```
## expression(cos(x + y))
```

```
eval(ex, list(x = pi, y = pi/2))
```

```
## [1] -1.83697e-16
```

```
## Previous syntax for Ryacas was
## x <- Sym("x")
## y <- Sym("y")
## dryacas <- deriv(sin(x+y), x)
## print(dryacas)
## class(dryacas)
```

```
detach("package:Ryacas", unload=TRUE)
```

Derivatives and simplifications – base R

See specific notes either in comments or at the end of the section.

The help page for `D` lists the functions for which derivatives are known: "The internal code knows about the arithmetic operators `+`, `-`, `*`, `/` and `^`, and the single-variable functions `exp`, `log`, `sin`, `cos`, `tan`, `sinh`, `cosh`, `sqrt`, `pnorm`, `dnorm`, `asin`, `acos`, `atan`, `gamma`, `lgamma`, `digamma` and `trigamma`, as well as `psigamma` for one or two arguments (but derivative only with respect to the first)."

Derivatives and simplifications – package `nlsr`

This package supports the derivatives that `D` supports, as well as a few others, and users can add their own definitions. The current list is

```
ls(nlsr::sysDerivs)
```

```
##  [1] "-"        "("        "*"        "/"        "^"        "+"
##  [7] "~"        "abs"      "acos"     "asin"     "atan"     "cos"
## [13] "cosh"     "digamma"  "dnorm"    "exp"      "gamma"    "lgamma"
## [19] "log"      "pnorm"    "psigamma" "sign"     "sin"      "sinh"
## [25] "sqrt"     "tan"      "trigamma"
```

Derivatives table

Here is a slightly expanded testing of the elements of the `nlsr` derivatives table.

```
require(nlsr)
```

```
## Loading required package: nlsr
```

```
## Try different ways to supply the log function
aDeriv <- nlsDeriv(~ log(x), "x")
class(aDeriv)
```

```
## [1] "call"
```

```
aDeriv
```

```
## 1/x
```

```
aderiv <- try(deriv( ~ log(x), "x"))
class(aderiv)
```

```
## [1] "expression"
```

```
aderiv
```

```
## expression({
##     .value <- log(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 1/x
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
aD <- D(expression(log(x)), "x")
class(aD)
```

```
## [1] "call"
```

```
aD
```

```
## 1/x
```

```
cat("but \n")
```

```
## but
```

```
try(D( "~ log(x)", "x")) # fails -- gives NA rather than expected answer due to quotes
```

```
## Error in D("~ log(x)", "x") : expression must not be type 'character'
```

```
try(D( ~ log(x), "x"))
```

```
## Error in D(~log(x), "x") : Function '`~`' is not in the derivatives table
```

```
interm <- ~ log(x)
interm
```

```
## ~log(x)
```

```
class(interm)
```

```
## [1] "formula"
```

```
interme <- as.expression(interm)
class(interme)
```

```
## [1] "expression"
```

```
try(D(interme, "x"))
```

```
## Error in D(interme, "x") : Function '`~`' is not in the derivatives table
```

```
try(deriv(interme, "x"))
```

```
## Error in deriv.default(interme, "x") :
##   Function '`~`' is not in the derivatives table
```

```
try(deriv(interm, "x"))
```

```
## expression({
##     .value <- log(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 1/x
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
nlsDeriv(~ log(x, base=3), "x" ) # OK
```

```
## 1/(x * 1.09861228866811)
```

```
try(D(expression(log(x, base=3)), "x" )) # fails - only single-argument calls supported
```

```
## Error in D(expression(log(x, base = 3)), "x") :
##   only single-argument calls to log() are supported;
##   maybe use log(x,a) = log(x)/log(a)
```

```
try(deriv(~ log(x, base=3), "x" )) # fails - only single-argument calls supported
```

```
## Error in deriv.formula(~log(x, base = 3), "x") :
##   only single-argument calls to log() are supported;
##   maybe use log(x,a) = log(x)/log(a)
```

```
try(deriv(expression(log(x, base=3)), "x" )) # fails - only single-argument calls supported
```

```
## Error in deriv.default(expression(log(x, base = 3)), "x") :
##   only single-argument calls to log() are supported;
##   maybe use log(x,a) = log(x)/log(a)
```

```
try(deriv3(expression(log(x, base=3)), "x" )) # fails - only single-argument calls supported
```

```
## Error in deriv3.default(expression(log(x, base = 3)), "x") :
##   only single-argument calls to log() are supported;
##   maybe use log(x,a) = log(x)/log(a)
```

```
fnDeriv(quote(log(x, base=3)), "x" )
```

```
## function (x)
## {
##     .value <- log(x, base = 3)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 1/(x * 1.09861228866811)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ exp(x), "x")
```

```
## exp(x)
```

```
D(expression(exp(x)), "x") # OK
```

```
## exp(x)
```

```
deriv(~exp(x), "x") # OK, but much more complicated
```

```
## expression({
##     .expr1 <- exp(x)
##     .value <- .expr1
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- .expr1
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(exp(x)), "x")
```

```
## function (x)
## {
##     .expr1 <- exp(x)
##     .value <- .expr1
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- .expr1
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ sin(x), "x")
```

```
## cos(x)
```

```
D(expression(sin(x)), "x")
```

```
## cos(x)
```

```
deriv(~sin(x), "x")
```

```
## expression({
##     .value <- sin(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- cos(x)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(sin(x)), "x")
```

```
## function (x)
## {
##     .value <- sin(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- cos(x)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ cos(x), "x")
```

```
## -sin(x)
```

```
D(expression(cos(x)), "x")
```

```
## -sin(x)
```

```
deriv(~ cos(x), "x")
```

```
## expression({
##     .value <- cos(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- -sin(x)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(cos(x)), "x")
```

```
## function (x)
## {
##     .value <- cos(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- -sin(x)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ tan(x), "x")
```

```
## 1/cos(x)^2
```

```
D(expression(tan(x)), "x")
```

```
## 1/cos(x)^2
```

```
deriv(~ tan(x), "x")
```

```
## expression({
##     .value <- tan(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 1/cos(x)^2
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(tan(x)), "x")
```

```
## function (x)
## {
##     .value <- tan(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 1/cos(x)^2
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ sinh(x), "x")
```

```
## cosh(x)
```

```
D(expression(sinh(x)), "x")
```

```
## cosh(x)
```

```
deriv(~sinh(x), "x")
```

```
## expression({
##     .value <- sinh(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- cosh(x)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(sinh(x)), "x")
```

```
## function (x)
## {
##     .value <- sinh(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- cosh(x)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ cosh(x), "x")
```

```
## sinh(x)
```

```
D(expression(cosh(x)), "x")
```

```
## sinh(x)
```

```
deriv(~cosh(x), "x")
```

```
## expression({
##     .value <- cosh(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- sinh(x)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(cosh(x)), "x")
```

```
## function (x)
## {
##     .value <- cosh(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- sinh(x)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ sqrt(x), "x")
```

```
## 0.5/sqrt(x)
```

```
D(expression(sqrt(x)), "x")
```

```
## 0.5 * x^-0.5
```

```
deriv(~sqrt(x), "x")
```

```
## expression({
##     .value <- sqrt(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 0.5 * x^-0.5
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(sqrt(x)), "x")
```

```
## function (x)
## {
##     .expr1 <- sqrt(x)
##     .value <- .expr1
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 0.5/.expr1
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ pnorm(q), "q")
```

```
## dnorm(q)
```

```
D(expression(pnorm(q)), "q")
```

```
## dnorm(q)
```

```
deriv(~pnorm(q), "q")
```

```
## expression({
##     .value <- pnorm(q)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("q")))
##     .grad[, "q"] <- dnorm(q)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(pnorm(q)), "q")
```

```
## function (q)
## {
##     .value <- pnorm(q)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "q"))
##     .grad[, "q"] <- dnorm(q)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ dnorm(x, mean), "mean")
```

```
## dnorm(x - mean) * (x - mean)
```

```
D(expression(dnorm(x, mean)), "mean")
```

```
## [1] 0
```

```
deriv(~dnorm(x, mean), "mean")
```

```
## expression({
##     .value <- dnorm(x, mean)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("mean")))
##     .grad[, "mean"] <- 0
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(dnorm(x, mean)), "mean")
```

```
## function (x, mean)
## {
##     .expr1 <- x - mean
##     .value <- dnorm(x, mean)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "mean"))
##     .grad[, "mean"] <- dnorm(.expr1) * .expr1
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ asin(x), "x")
```

```
## 1/sqrt(1 + x^2)
```

```
D(expression(asin(x)), "x")
```

```
## 1/sqrt(1 - x^2)
```

```
deriv(~asin(x), "x")
```

```
## expression({
##     .value <- asin(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 1/sqrt(1 - x^2)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(asin(x)), "x")
```

```
## function (x)
## {
##     .value <- asin(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 1/sqrt(1 + x^2)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ acos(x), "x")
```

```
## -1/sqrt(1 + x^2)
```

```
D(expression(acos(x)), "x")
```

```
## -(1/sqrt(1 - x^2))
```

```
deriv(~acos(x), "x")
```

```
## expression({
##     .value <- acos(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- -(1/sqrt(1 - x^2))
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(acos(x)), "x")
```

```
## function (x)
## {
##     .value <- acos(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- -1/sqrt(1 + x^2)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ atan(x), "x")
```

```
## 1/(1 + x^2)
```

```
D(expression(atan(x)), "x")
```

```
## 1/(1 + x^2)
```

```
deriv(~atan(x), "x")
```

```
## expression({
##     .value <- atan(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 1/(1 + x^2)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(atan(x)), "x")
```

```
## function (x)
## {
##     .value <- atan(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 1/(1 + x^2)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ gamma(x), "x")
```

```
## gamma(x) * digamma(x)
```

```
D(expression(gamma(x)), "x")
```

```
## gamma(x) * digamma(x)
```

```
deriv(~gamma(x), "x")
```

```
## expression({
##     .expr1 <- gamma(x)
##     .value <- .expr1
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- .expr1 * digamma(x)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(gamma(x)), "x")
```

```
## function (x)
## {
##     .expr1 <- gamma(x)
##     .value <- .expr1
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- .expr1 * digamma(x)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ lgamma(x), "x")
```

```
## digamma(x)
```

```
D(expression(lgamma(x)), "x")
```

```
## digamma(x)
```

```
deriv(~lgamma(x), "x")
```

```
## expression({
##     .value <- lgamma(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- digamma(x)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(lgamma(x)), "x")
```

```
## function (x)
## {
##     .value <- lgamma(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- digamma(x)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ digamma(x), "x")
```

```
## trigamma(x)
```

```
D(expression(digamma(x)), "x")
```

```
## trigamma(x)
```

```
deriv(~digamma(x), "x")
```

```
## expression({
##     .value <- digamma(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- trigamma(x)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(digamma(x)), "x")
```

```
## function (x)
## {
##     .value <- digamma(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- trigamma(x)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ trigamma(x), "x")
```

```
## psigamma(x, 2L)
```

```
D(expression(trigamma(x)), "x")
```

```
## psigamma(x, 2L)
```

```
deriv(~trigamma(x), "x")
```

```
## expression({
##     .value <- trigamma(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- psigamma(x, 2L)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(trigamma(x)), "x")
```

```
## function (x)
## {
##     .value <- trigamma(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- psigamma(x, 2L)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ psigamma(x, deriv = 5), "x")
```

```
## psigamma(x, 6)
```

```
D(expression(psigamma(x, deriv = 5)), "x")
```

```
## psigamma(x, 6L)
```

```
deriv(~psigamma(x, deriv = 5), "x")
```

```
## expression({
##     .value <- psigamma(x, deriv = 5)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- psigamma(x, 6L)
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(psigamma(x, deriv = 5)), "x")
```

```
## function (x)
## {
##     .value <- psigamma(x, deriv = 5)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- psigamma(x, 6)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ x*y, "x")
```

```
## y
```

```
D(expression(x*y), "x")
```

```
## y
```

```
deriv(~x*y, "x")
```

```
## expression({
##     .value <- x * y
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- y
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(x*y), "x")
```

```
## function (x, y)
## {
##     .value <- x * y
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- y
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ x/y, "x")
```

```
## 1/y
```

```
D(expression(x/y), "x")
```

```
## 1/y
```

```
deriv(~x/y, "x")
```

```
## expression({
##     .value <- x/y
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 1/y
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(x/y), "x")
```

```
## function (x, y)
## {
##     .value <- x/y
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 1/y
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ x^y, "x")
```

```
## y * x^(y - 1)
```

```
D(expression(x^y), "x")
```

```
## x^(y - 1) * y
```

```
deriv(~x^y, "x")
```

```
## expression({
##     .value <- x^y
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- x^(y - 1) * y
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(x^y), "x")
```

```
## function (x, y)
## {
##     .value <- x^y
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- y * x^(y - 1)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ (x), "x")
```

```
## [1] 1
```

```
D(expression((x)), "x")
```

```
## [1] 1
```

```
deriv(~(x), "x")
```

```
## expression({
##     .value <- (x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 1
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote((x)), "x")
```

```
## function (x)
## {
##     .value <- (x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 1
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ +x, "x")
```

```
## [1] 1
```

```
D(expression(+x), "x")
```

```
## [1] 1
```

```
deriv(~ +x, "x")
```

```
## expression({
##     .value <- +x
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- 1
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(+x), "x")
```

```
## function (x)
## {
##     .value <- +x
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 1
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ -x, "x")
```

```
## [1] -1
```

```
D(expression(- x), "x")
```

```
## -1
```

```
deriv(~ -x, "x")
```

```
## expression({
##     .value <- -x
##     .grad <- array(0, c(length(.value), 1L), list(NULL, c("x")))
##     .grad[, "x"] <- -1
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
fnDeriv(quote(-x), "x")
```

```
## function (x)
## {
##     .value <- -x
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- -1
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ abs(x), "x")
```

```
## sign(x)
```

```
try(D(expression(abs(x)), "x")) # 'abs' not in derivatives table
```

```
## Error in D(expression(abs(x)), "x") :
##   Function 'abs' is not in the derivatives table
```

```
try(deriv(~ abs(x), "x"))
```

```
## Error in deriv.formula(~abs(x), "x") :
##   Function 'abs' is not in the derivatives table
```

```
fnDeriv(quote(abs(x)), "x")
```

```
## function (x)
## {
##     .value <- abs(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- sign(x)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
nlsDeriv(~ sign(x), "x")
```

```
## [1] 0
```

```
try(D(expression(sign(x)), "x")) # 'sign' not in derivatives table
```

```
## Error in D(expression(sign(x)), "x") :
##   Function 'sign' is not in the derivatives table
```

```
try(deriv(~ sign(x), "x"))
```

```
## Error in deriv.formula(~sign(x), "x") :
##   Function 'sign' is not in the derivatives table
```

```
fnDeriv(quote(sign(x)), "x")
```

```
## function (x)
## {
##     .value <- sign(x)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 0
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

Notes:

- the base tool `deriv` (and `deriv3`) and `nlsr::codeDeriv` are intended to output an expression to compute a derivative. `deriv` generates an expression object, while `codeDeriv` will generate a language object. Note that input to `deriv` is of the form of a tilde expression with no left hand side, while `codeDeriv` is more flexible: quoted expressions, or length-1 expression vectors may also be used.
- the base tool `D` and `nlsr::nlsDeriv` generate expressions, but `D` requires an expression, while `nlsDeriv` can handle the expression without a wrapper. ?? Do we need to discuss more??

- **nlsr** includes `abs(x)` and `sign(x)` in the derivatives table despite conventional wisdom that these are not differentiable. However, `abs(x)` clearly has a defined derivative everywhere except at x = 0, where assigning a value of 0 to the derivative is almost certainly acceptable in computations. Similarly for `sign(x)`.

Simplifying algebraic expressions

nlsr also includes some tools for simplification of algebraic expressions, extensible by the user. Currently these involve the following functions:

```
ls(nlsr::sysSimplifications)
```

```
##  [1] "-"       "!"       "("       "*"       "/"       "&&"      "^"
##  [8] "+"       "||"      "exp"     "if"      "log"     "missing"
```

```
#- Remove ##? to see reproducible error
#- ?? For some reason, if we leave packages attached, we get errors.
#- Here we detach all the non-base packages and then reload nlsr
##? require(nlsr)
##? sessionInfo()
##? ##? nlsSimplify(quote(+(a+b)))
##? nlsSimplify(quote(-5))
```

```
#- ?? For some reason, if we leave packages attached, we get errors.
#- Here we detach all the non-base packages and then reload nlsr
sessionInfo()
```

```
## R version 3.6.1 (2019-07-05)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Linux Mint 19.1
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/openblas/libblas.so.3
## LAPACK: /usr/lib/x86_64-linux-gnu/libopenblasp-r0.2.20.so
##
## locale:
##  [1] LC_CTYPE=en_CA.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_CA.UTF-8        LC_COLLATE=en_CA.UTF-8
##  [5] LC_MONETARY=en_CA.UTF-8    LC_MESSAGES=en_CA.UTF-8
##  [7] LC_PAPER=en_CA.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_CA.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] nlsr_2019.8.21
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.2       rprojroot_1.3-2  crayon_1.3.4     assertthat_0.2.1
##  [5] withr_2.1.2      digest_0.6.20    R6_2.4.0         backports_1.1.4
##  [9] magrittr_1.5     evaluate_0.14    rlang_0.4.0      stringi_1.4.3
## [13] rstudioapi_0.10  testthat_2.2.1   rmarkdown_1.15   desc_1.2.0
## [17] tools_3.6.1      stringr_1.4.0    xfun_0.9         pkgload_1.0.2
```

```
## [21] yaml_2.2.0       compiler_3.6.1   htmltools_0.3.6  knitr_1.24
```

```
if ("Deriv" %in% loadedNamespaces()){detach("package:Deriv", unload=TRUE)}
  #- ?? Do we need to unload too.
if ("nlsr" %in% loadedNamespaces() ){detach("package:nlsr", unload=TRUE)}
if ("Ryacas" %in% loadedNamespaces() ){detach("package:Ryacas", unload=TRUE)}
require(nlsr)
```

```
## Loading required package: nlsr
```

```
#- require(Deriv)
#- require(stats)
#- Various simplifications
#- ?? Do we need quote() to stop attempt to evaluate before applying simplification

nlsSimplify(quote(+(a+b)))
```

```
## a + b
```

```
nlsSimplify(quote(-5))
```

```
## [1] -5
```

```
nlsSimplify(quote(--(a+b)))
```

```
## a + b
```

```
nlsSimplify(quote(exp(log(a+b))))
```

```
## a + b
```

```
nlsSimplify(quote(exp(1)))
```

```
## [1] 2.718282
```

```
nlsSimplify(quote(log(exp(a+b))))
```

```
## a + b
```

```
nlsSimplify(quote(log(1)))
```

```
## [1] 0
```

```
nlsSimplify(quote(!TRUE))
```

```
## [1] FALSE
```

```
nlsSimplify(quote(!FALSE))
```

```
## [1] TRUE
```

```
nlsSimplify(quote((a+b)))
```

```
## a + b
```

```
nlsSimplify(quote(a + b + 0))
```

```
## a + b
```

```
nlsSimplify(quote(0 + a + b))
```

```
## a + b
```

```
nlsSimplify(quote((a+b) + (a+b)))
```

```
## 2 * (a + b)
```

```
nlsSimplify(quote(1 + 4))
```

```
## [1] 5
```

```
nlsSimplify(quote(a + b - 0))
```

```
## a + b
```

```
nlsSimplify(quote(0 - a - b))
```

```
## -a - b
```

```
nlsSimplify(quote((a+b) - (a+b)))
```

```
## [1] 0
```

```
nlsSimplify(quote(5 - 3))
```

```
## [1] 2
```

```
nlsSimplify(quote(0*(a+b)))
```

```
## [1] 0
```

```
nlsSimplify(quote((a+b)*0))
```

```
## [1] 0
```

```
nlsSimplify(quote(1L * (a+b)))
```

```
## a + b
```

```
nlsSimplify(quote((a+b) * 1))
```

```
## a + b
```

```
nlsSimplify(quote((-1)*(a+b)))
```

```
## -(a + b)
```

```
nlsSimplify(quote((a+b)*(-1)))
```

```
## -(a + b)
```

```
nlsSimplify(quote(2*5))
```

```
## [1] 10
```

```
nlsSimplify(quote((a+b) / 1))
```

```
## a + b
```

```
nlsSimplify(quote((a+b) / (-1)))
```

```
## -(a + b)
```

```
nlsSimplify(quote(0/(a+b)))
```

```
## [1] 0
```

```
nlsSimplify(quote(1/3))
```

```
## [1] 0.3333333
```

```
nlsSimplify(quote((a+b) ^ 1))
```

```
## a + b
```

```
nlsSimplify(quote(2^10))
```

```
## [1] 1024
```

```
nlsSimplify(quote(log(exp(a), 3)))
```

```
## a/1.09861228866811
```

```
nlsSimplify(quote(FALSE && b))
```

```
## [1] FALSE
```

```
nlsSimplify(quote(a && TRUE))
```

```
## a
```

```
nlsSimplify(quote(TRUE && b))
```

```
## b
```

```
nlsSimplify(quote(a || TRUE))
```

```
## [1] TRUE
```

```
nlsSimplify(quote(FALSE || b))
```

```
## b
```

```
nlsSimplify(quote(a || FALSE))
```

```
## a
```

```
nlsSimplify(quote(if (TRUE) a+b))
```

```
## a + b
```

```
nlsSimplify(quote(if (FALSE) a+b))
```

```
## NULL
```

```
nlsSimplify(quote(if (TRUE) a+b else a*b))
```

```
## a + b
```

```
nlsSimplify(quote(if (FALSE) a+b else a*b))
```

```
## a * b
```

```
nlsSimplify(quote(if (cond) a+b else a+b))
```

```
## a + b
```

```
nlsSimplify(quote(--(a+b)))
```

```
## a + b
```

```
nlsSimplify(quote(-(-(a+b))))
```

```
## a + b
```

Derivatives and simplifications – package `Deriv`

Derivatives table

Simplifications

```
#- ?? For some reason, if we leave packages attached, we get errors.
#- Here we detach all the non-base packages and then reload nlsr
sessionInfo()
```

```
## R version 3.6.1 (2019-07-05)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Linux Mint 19.1
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/openblas/libblas.so.3
## LAPACK: /usr/lib/x86_64-linux-gnu/libopenblasp-r0.2.20.so
##
## locale:
##  [1] LC_CTYPE=en_CA.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_CA.UTF-8        LC_COLLATE=en_CA.UTF-8
##  [5] LC_MONETARY=en_CA.UTF-8    LC_MESSAGES=en_CA.UTF-8
##  [7] LC_PAPER=en_CA.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_CA.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] nlsr_2019.8.21
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.2       rprojroot_1.3-2  crayon_1.3.4     assertthat_0.2.1
##  [5] withr_2.1.2      digest_0.6.20    R6_2.4.0         backports_1.1.4
##  [9] magrittr_1.5     evaluate_0.14    rlang_0.4.0      stringi_1.4.3
## [13] rstudioapi_0.10  testthat_2.2.1   rmarkdown_1.15   desc_1.2.0
## [17] tools_3.6.1      stringr_1.4.0    xfun_0.9         pkgload_1.0.2
## [21] yaml_2.2.0       compiler_3.6.1   htmltools_0.3.6  knitr_1.24
```

```
if ("Deriv" %in% loadedNamespaces()){detach("package:Deriv", unload=TRUE)}
  #- ?? Do we need to unload too.
if ("Deriv" %in% loadedNamespaces() ){detach("package:nlsr", unload=TRUE)}
if ("Deriv" %in% loadedNamespaces() ){detach("package:Ryacas", unload=TRUE)}
require(Deriv)
```

```
## Loading required package: Deriv
```

```
##
## Attaching package: 'Deriv'
```

```
## The following object is masked _by_ '.GlobalEnv':
##
##     drule
```

```
#- Various simplifications
#- ?? Do we need quote() to stop attempt to evaluate before applying simplification

Simplify(quote(+(a+b)))
```

```
## a + b
```

```
Simplify(quote(-5))
```

```
## [1] -5
```

```
Simplify(quote(--(a+b)))
```

```
## a + b
```

```
Simplify(quote(exp(log(a+b))))
```

```
## exp(log(a + b))
```

```
Simplify(quote(exp(1)))
```

```
## [1] 2.718282
```

```
Simplify(quote(log(exp(a+b))))
```

```
## a + b
```

```
Simplify(quote(log(1)))
```

```
## [1] 0
```

```
Simplify(quote(!TRUE))
```

```
## [1] FALSE
```

```
Simplify(quote(!FALSE))
```

```
## [1] TRUE
```

```
Simplify(quote((a+b)))
```

```
## a + b
```

```
Simplify(quote(a + b + 0))
```

```
## a + b
```

```
Simplify(quote(0 + a + b))
```

```
## a + b
```

```
Simplify(quote((a+b) + (a+b)))
```

```
## 2 * (a + b)
```

```
Simplify(quote(1 + 4))
```

```
## [1] 5
```

```
Simplify(quote(a + b - 0))
```

```
## a + b
Simplify(quote(0 - a - b))

## -(a + b)
Simplify(quote((a+b) - (a+b)))

## [1] 0
Simplify(quote(5 - 3))

## [1] 2
Simplify(quote(0*(a+b)))

## [1] 0
Simplify(quote((a+b)*0))

## [1] 0
Simplify(quote(1L * (a+b)))

## a + b
Simplify(quote((a+b) * 1))

## a + b
Simplify(quote((-1)*(a+b)))

## -(a + b)
Simplify(quote((a+b)*(-1)))

## -(a + b)
Simplify(quote(2*5))

## [1] 10
Simplify(quote((a+b) / 1))

## a + b
Simplify(quote((a+b) / (-1)))

## -(a + b)
Simplify(quote(0/(a+b)))

## [1] 0
Simplify(quote(1/3))

## [1] 0.3333333
Simplify(quote((a+b) ^ 1))

## a + b
Simplify(quote(2^10))

## [1] 1024
```

```
Simplify(quote(log(exp(a), 3)))
```

```
## a/1.09861228866811
```

```
Simplify(quote(FALSE && b))
```

```
## FALSE && b
```

```
Simplify(quote(a && TRUE))
```

```
## a && TRUE
```

```
Simplify(quote(TRUE && b))
```

```
## TRUE && b
```

```
Simplify(quote(a || TRUE))
```

```
## a || TRUE
```

```
Simplify(quote(FALSE || b))
```

```
## FALSE || b
```

```
Simplify(quote(a || FALSE))
```

```
## a || FALSE
```

```
Simplify(quote(if (TRUE) a+b))
```

```
## a + b
```

```
Simplify(quote(if (FALSE) a+b))
```

```
## if (FALSE) a + b
```

```
Simplify(quote(if (TRUE) a+b else a*b))
```

```
## a + b
```

```
Simplify(quote(if (FALSE) a+b else a*b))
```

```
## a * b
```

```
Simplify(quote(if (cond) a+b else a+b))
```

```
## a + b
```

```
#- This one is wrong... the double minus is an error, yet it works ??.
Simplify(quote(--(a+b)))
```

```
## a + b
```

```
#- By comparison
Simplify(quote(-(-(a+b))))
```

```
## a + b
```

Comparison with other approaches

check modelexpr() works with an ssgrfun ??

test model2rjfun vs model2rjfunx ??

Need more extensive discussion of Simplify??

Issues of programming on the language

?? need to explain where Deriv package comes from

One of the key tasks with tools for derivatives is that of taking objects in one or other form (that is, **R** class) and using it as an input for a symbolic function. The object may, of course, be an output from another such function, and this is one of the reasons we need to do such transformations.

We also note that the different tools for symbolic derivatives use slightly different inputs. For example, for the derivative of log(x), we have

```
require(nlsr)
dlogx <- nlsr::nlsDeriv(~ log(x), "x")
str(dlogx)
```

```
##  language 1/x
```

```
print(dlogx)
```

```
## 1/x
```

Unfortunately, there are complications when we have an expression object, and we need to specify that we do NOT execute the *substitute()* function. Here we show how to do this implicitly and with an explicit object.

```
require(nlsr)
dlogxs <- nlsr::nlsDeriv(expression(log(x)), "x", do_substitute=FALSE)
str(dlogxs)
```

```
##  language 1/x
```

```
print(dlogxs)
```

```
## 1/x
```

```
cat(as.character(dlogxs), "\n")
```

```
## / 1 x
```

```
fne <- expression(log(x))
dlogxe <- nlsr::nlsDeriv(fne, "x", do_substitute=FALSE)
str(dlogxe)
```

```
##  language 1/x
```

```
print(dlogxe)
```

```
## 1/x
```

```
# base R
dblogx <- D(expression(log(x)), "x")
str(dblogx)
```

```
##  language 1/x
```

```
print(dblogx)
```

```
## 1/x
```

```
require(Deriv)
ddlogx <- Deriv::Deriv(expression(log(x)), "x")
str(ddlogx)
```

```
##   expression(1/x)
```

```
print(ddlogx)
```

```
## expression(1/x)
```

```
cat(as.character(ddlogx), "\n")
```

```
## 1/x
```

```
ddlogxf <- ~ ddlogx
str(ddlogxf)
```

```
## Class 'formula'  language ~ddlogx
##   ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
```

??? do each example by all methods and by numDeriv and put in dataframe for later presentation in a table.

Do we want examples in columns or rows. Probably 1 fn per row and work out a name for the row that is reasonably meaningful. Probably want an index column as well that is a list of strings. Can we then act on those strings to automate the whole setup?

```
require(nlsr)
# require(stats)
# require(Deriv)
# require(Ryacas)

# Various derivatives

new <- codeDeriv(quote(1 + x + y), c("x", "y"))
old <- deriv(quote(1 + x + y), c("x", "y"))
print(new)
```

```
## {
##     .value <- 1 + x + y
##     .grad <- array(0, c(length(.value), 2L), list(NULL, c("x",
##     "y")))
##     .grad[, "x"] <- 1
##     .grad[, "y"] <- 1
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
# Following generates a very long line on output of knitr (for markdown)
class(new)
```

```
## [1] "{"
```

```
str(new)
```

```
##  language {  .value <- 1 + x + y; .grad <- array(0, c(length(.value), 2L), list(NULL, c("x", "y"; ))
```

```
as.expression(new)
```

```
## expression({
##     .value <- 1 + x + y
##     .grad <- array(0, c(length(.value), 2L), list(NULL, c("x",
##     "y")))
##     .grad[, "x"] <- 1
##     .grad[, "y"] <- 1
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
newf <- function(x, y){
   eval(new)
}
newf(3,5)
```

```
## [1] 9
## attr(,"gradient")
##      x y
## [1,] 1 1
```

```
print(old)
```

```
## expression({
##     .value <- 1 + x + y
##     .grad <- array(0, c(length(.value), 2L), list(NULL, c("x",
##         "y")))
##     .grad[, "x"] <- 1
##     .grad[, "y"] <- 1
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
class(old)
```

```
## [1] "expression"
```

```
str(old)
```

```
##   expression({  .value <- 1 + x + y  .grad <- array(0, c(length(.value), 2L), list(NULL, c("x", "y")))
```

```
oldf <- function(x,y){
    eval(old)
}
oldf(3,5)
```

```
## [1] 9
## attr(,"gradient")
##      x y
## [1,] 1 1
```

Unfortunately, the inputs and outputs are not always easily transformed so that the symbolic derivatives can be found. (?? Need to codify this and provide filters so we can get things to work nicely.)

As an example, how could we take object **new** and embed it in a function we can then use in **R**? We can certainly copy and paste the output into a function template, as follows,

```
fnfromnew <- function(x,y){
    .value <- 1 + x + y
    .grad <- array(0, c(length(.value), 2L), list(NULL, c("x",
    "y")))
    .grad[, "x"] <- 1
    .grad[, "y"] <- 1
    attr(.value, "gradient") <- .grad
    .value
}

print(fnfromnew(3,5))
```

```
## [1] 9
## attr(,"gradient")
##      x y
## [1,] 1 1
```

However, we would ideally like to be able to automate this to generate functions and gradients for nonlinear least squares and optimization calculations. The same criticism applies to the object **old**

####Another issue:

If we have x and y set such that the function is not admissible, then both our old and new functions give a gradient that is seemingly reasonable. While the gradient of this simple function could be considered to be defined for ANY values of x and y, I (JN) am sure most users would wish for a warning at the very least in such cases.

```
x <- NA
y <- Inf
print(eval(new))
```

```
## [1] NA
## attr(,"gradient")
##      x y
## [1,] 1 1
```

```
print(eval(old))
```

```
## [1] NA
## attr(,"gradient")
##      x y
## [1,] 1 1
```

####SafeD

We could define a way to avoid the issue of character vs. expression (and possibly other classes) as follows:

```
safeD <- function(obj, var) {
   # safeguarded D() function for symbolic derivs
   if (! is.character(var) ) stop("The variable var MUST be character type")
   if (is.character(obj) ) {
       eobj <- parse(text=obj)
       result <- D(eobj, var)
   } else {
       result <- D(obj, var)
   }
}
```

```
lxy2 <- expression(log(x+y^2))
clxy2 <- "log(x+y^2)"
try(print(D(clxy2, "y")))
```

```
## Error in D(clxy2, "y") : expression must not be type 'character'
```

```
print(try(D(lxy2, "y")))
```

```
## 2 * y/(x + y^2)
```

```
print(safeD(clxy2, "y"))
```

```
## 2 * y/(x + y^2)
```

```
print(safeD(lxy2, "y"))
```

```
## 2 * y/(x + y^2)
```

Indexed parameters or variables

Erin Hodgess on R-help in January 2015 raised the issue of taking the derivative of an expression that contains an indexed variable. We show the example and its resolution, then give an explanation.

```
zzz <- expression(y[3]*r1 + r2)
try(deriv(zzz,c("r1","r2")))
```

```
## Error in deriv.default(zzz, c("r1", "r2")) :
##   Function '`[`' is not in the derivatives table
```

```
require(nlsr)
try(nlsr::nlsDeriv(zzz, c("r1","r2")))
```

```
## Error in nlsDeriv(expr[[2]], name, derivEnv, do_substitute = FALSE, verbose = verbose,  :
##   no derivative known for '['
```

```
try(fnDeriv(zzz, c("r1","r2")))
```

```
## Error in nlsDeriv(expr[[2]], name, derivEnv, do_substitute = FALSE, verbose = verbose,  :
##   no derivative known for '['
```

```
newDeriv(`[`(x,y), stop("no derivative when indexing"))
try(nlsr::nlsDeriv(zzz, c("r1","r2")))
```

```
## y[3] * c(1, 0) + stop("no derivative when indexing") * r1 + c(0,
## 1)
```

```
try(nlsr::fnDeriv(zzz, c("r1","r2")))
```

```
## function (y, r1, r2)
## {
##     .expr1 <- y[3]
##     .expr2 <- stop("no derivative when indexing")
##     .expr3 <- .expr2 * r1
##     .value <- .expr1 * r1 + r2
##     .grad <- array(0, c(length(.value), 2L), list(NULL, c("r1",
##     "r2")))
##     .grad[, "r1"] <- .expr1 + .expr3
##     .grad[, "r2"] <- .expr3 + 1
##     attr(.value, "gradient") <- .grad
```

```
##     .value
## }
```

Richard Heiberger pointed out that internally, **R** stores

`y[3]`

as

`"["(y,3)`

that is, as a function. Duncan Murdoch pointed out the availability of **nlsr** and the use of newDeriv() to redefine the "[" function for the purposes of derivatives.

This is not an ideal resolution, especially as we would like to be able to get the gradients of functions with respect to vectors of parameters (Noted also by Sergei Sokol in the manual for package **Deriv**). The following examples illustrate this.

```
try(nlsr::nlsDeriv(zzz, "y[3]"))
```

```
## stop("no derivative when indexing") * r1
```

```
try(nlsr::nlsDeriv(y3*r1+r2,"y3"))
```

```
## Error : object 'y3' not found
```

```
try(nlsr::nlsDeriv(y[3]*r1+r2,"y[3]"))
```

```
## Error : object 'r1' not found
```

References

Clausen, Andrew, and Serguei Sokol. 2018. *Deriv: R-Based Symbolic Differentiation*. https://CRAN.R-project.org/package=Deriv.

Goedman, Rob, Gabor Grothendieck, Søren Højsgaard, Ayal Pinkus, Grzegorz Mazur, and Mikkel Meyer Andersen. 2019. *Ryacas: R Interface to the Yacas Computer Algebra System*. https://CRAN.R-project.org/package=Ryacas.

Nash, John C, and Duncan Murdoch. 2019. *Nlsr: Functions for Nonlinear Least Squares Solutions*.

R Development Core Team. 2008. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. http://www.R-project.org.