Symbolic and analytical derivatives in R

John C. Nash

2023-05-09

Contents

This article is an attempt to catalog and illustrate the various capabilities in the **R** statistical computing system to perform analytic or symbolic 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`, and `Ryacas`. General tools for approximations to derivatives are found in the package `numDeriv` as well as `optimx`. Other 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**, I have attempted to put comments in the **R** code with the preface '`#-`'.)

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 (formerly `nlsr`) `nlsDeriv()`, `fnDeriv()`, and the wrapper `model2rjfun` (John C Nash and Duncan 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).

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. There is an example below on the section for `Deriv`.

stats (i.e., the base R installation)

`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.

```
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
## })
```

```
mode(dx2x)
```

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

```
str(dx2x)
```

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

```
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
```

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

```
## function (object, ...)
## UseMethod("str")
## <bytecode: 0x5635f26bd3d8>
## <environment: namespace:utils>
```

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

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

```
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"
```

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

```
## [1] 1
## attr(,"gradient")
##      x
## [1,] 2
```

```
fdx2x(3.21)
```

```
## [1] 10.304
## 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
```

```
#-  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))
```

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

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

```
( 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
## })
```

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

```
## [1]  0.84147  0.51440 -0.40424 -0.83602
## attr(,"gradient")
##              x        y
## [1,]  0.000000  0.00000
## [2,] -0.721606 -1.44321
## [3,] -0.831692 -1.66338
## [4,] -0.077432 -0.15486
```

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

```
## [1]  0.000000 -0.721606 -0.831692 -0.077432
```

```
#-  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] 4.5227 4.1213 3.7838 3.5000 3.2613 3.0607 2.8919
## attr(,"gradient")
##      b0      b1      th
## [1,]  1 0.84090 0.10929
## [2,]  1 0.70711 0.18380
## [3,]  1 0.59460 0.23183
## [4,]  1 0.50000 0.25993
## [5,]  1 0.42045 0.27322
## [6,]  1 0.35355 0.27570
## [7,]  1 0.29730 0.27047
```

```
#-  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] 4.5227 4.1213 3.7838 3.5000 3.2613 3.0607 2.8919
## attr(,"gradient")
##      b0      b1      th
## [1,]  1 0.84090 0.10929
## [2,]  1 0.70711 0.18380
## [3,]  1 0.59460 0.23183
## [4,]  1 0.50000 0.25993
## [5,]  1 0.42045 0.27322
## [6,]  1 0.35355 0.27570
## [7,]  1 0.29730 0.27047
## 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 0.036429
## [2,]  0  0 0.061266
## [3,]  0  0 0.077278
## [4,]  0  0 0.086643
## [5,]  0  0 0.091073
## [6,]  0  0 0.091899
## [7,]  0  0 0.090157
##
## , , th
##
##      b0       b1        th
## [1,]  0 0.036429 -0.049909
## [2,]  0 0.061266 -0.075974
## [3,]  0 0.077278 -0.085786
## [4,]  0 0.086643 -0.084923
## [5,]  0 0.091073 -0.077428
## [6,]  0 0.091899 -0.066187
## [7,]  0 0.090157 -0.053215
```

```
#- 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] 4.5227 4.1213 3.7838 3.5000 3.2613 3.0607 2.8919
## attr(,"gradient")
##      b0      b1      th
## [1,]  1 0.84090 0.10929
## [2,]  1 0.70711 0.18380
## [3,]  1 0.59460 0.23183
## [4,]  1 0.50000 0.25993
## [5,]  1 0.42045 0.27322
## [6,]  1 0.35355 0.27570
## [7,]  1 0.29730 0.27047
## 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 0.036429
```

```
## [2,]  0  0 0.061266
## [3,]  0  0 0.077278
## [4,]  0  0 0.086643
## [5,]  0  0 0.091073
## [6,]  0  0 0.091899
## [7,]  0  0 0.090157
##
## , , th
##
##      b0       b1        th
## [1,]  0 0.036429 -0.049909
## [2,]  0 0.061266 -0.075974
## [3,]  0 0.077278 -0.085786
## [4,]  0 0.086643 -0.084923
## [5,]  0 0.091073 -0.077428
## [6,]  0 0.091899 -0.066187
## [7,]  0 0.090157 -0.053215
```

```
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

```
library(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
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
## 2 * x
```

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

```
## 2 * x
```

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

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

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

```
##  num 0
```

```r
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] 2.1892
## attr(,"gradient")
##      b0     b1      th
## [1,]  1 0.5946 0.15456
```

```
#- 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)
```

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

```
## [1] 2.1892
## attr(,"gradient")
##      b0     b1      th
## [1,]  1 0.5946 0.15456
```

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

```
## [1] 2.1892
## attr(,"gradient")
##      b0     b1      th
## [1,]  1 0.5946 0.15456
```

```
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
## }
```

```
mode(dx2xnf)
```

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

```
str(dx2xnf)
```

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

```
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
## 0.6471 0.1068
```

```
#- 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] -0.45465  0.00000  0.45465  0.49130
```

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

```
## [1]  1.4161  0.0000 -1.4161 -1.5303
```

```
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
## 0.94808 0.32503
```

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

```
##       x       y
```

```
## 0.94808 0.32503
```

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

```
##        x        y
## -0.35317  2.54731
```

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

```
##       x       y
## 0.94808 0.32503
```

```
#-         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) < 0.1) 10 * x 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] 0.01
```

```
#- 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")
```

```
## NULL
```

```
  # 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
##  0.38528  0.18301
```

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
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.

```
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] -1.837e-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"       "cosh"
## [13] "digamma"   "dnorm"     "exp"       "gamma"     "lgamma"    "log"
## [19] "pnorm"     "psigamma"  "sign"      "sin"       "sinh"      "sqrt"
## [25] "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
##? 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 4.3.0 (2023-04-21)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Linux Mint 21.1
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
## LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.20.so;  LAPACK version 3.10.0
##
## locale:
##  [1] LC_CTYPE=en_CA.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_CA.UTF-8        LC_COLLATE=C
##  [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
##
## time zone: America/Toronto
## tzcode source: system (glibc)
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] nlsr_2023.5.8    minpack.lm_1.2-3 ggplot2_3.4.2    nlraa_1.5
## [5] optimx_2022-4.30
##
## loaded via a namespace (and not attached):
##  [1] Matrix_1.5-4        gtable_0.3.3        highr_0.10
##  [4] dplyr_1.0.8         compiler_4.3.0      Rcpp_1.0.10
##  [7] tinytex_0.45        tidyselect_1.1.2    assertthat_0.2.1
## [10] splines_4.3.0       scales_1.2.1        boot_1.3-28.1
## [13] yaml_2.3.7          fastmap_1.1.1       lattice_0.21-8
## [16] R6_2.5.1            generics_0.1.2      knitr_1.42
## [19] MASS_7.3-59         tibble_3.2.1        munsell_0.5.0
## [22] DBI_1.1.3           pillar_1.9.0        rlang_1.1.0
## [25] utf8_1.2.3          xfun_0.39           pkgload_1.3.2
## [28] cli_3.6.1           withr_2.5.0         magrittr_2.0.3
## [31] mgcv_1.8-42         digest_0.6.31       grid_4.3.0
## [34] lifecycle_1.0.3     nlme_3.1-162        vctrs_0.6.2
## [37] evaluate_0.20       glue_1.6.2          numDeriv_2016.8-1.1
## [40] fansi_1.0.4         colorspace_2.1-0    rmarkdown_2.21
## [43] purrr_0.3.4         tools_4.3.0         pkgconfig_2.0.3
## [46] htmltools_0.5.5
```

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

```r
if ("Ryacas" %in% loadedNamespaces() ){detach("package:Ryacas", unload=TRUE)}
#- require(Deriv)
#- require(stats)
#- Various simplifications
#- ?? Do we need quote() to stop attempt to evaluate before applying simplification
require(nlsr)
```

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

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

```
## a + b
```

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

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

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

```
## a + b
```

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

```
## a + b
```

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

```
## [1] 2.7183
```

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

```
## a + b
```

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

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

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

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

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

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

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

```
## a + b
```

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

```
## a + b
```

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

```
## a + b
```

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

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

```r
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] 0.33333
```

```
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 4.3.0 (2023-04-21)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Linux Mint 21.1
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/openblas-pthread/libblas.so.3
## LAPACK: /usr/lib/x86_64-linux-gnu/openblas-pthread/libopenblasp-r0.3.20.so;  LAPACK version 3.10.0
##
## locale:
##  [1] LC_CTYPE=en_CA.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_CA.UTF-8        LC_COLLATE=C
##  [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
##
## time zone: America/Toronto
## tzcode source: system (glibc)
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] nlsr_2023.5.8    minpack.lm_1.2-3 ggplot2_3.4.2    nlraa_1.5
## [5] optimx_2022-4.30
##
## loaded via a namespace (and not attached):
##  [1] Matrix_1.5-4        gtable_0.3.3        highr_0.10
##  [4] dplyr_1.0.8         compiler_4.3.0      Rcpp_1.0.10
##  [7] tinytex_0.45        tidyselect_1.1.2    assertthat_0.2.1
## [10] splines_4.3.0       scales_1.2.1        boot_1.3-28.1
## [13] yaml_2.3.7          fastmap_1.1.1       lattice_0.21-8
## [16] R6_2.5.1            generics_0.1.2      knitr_1.42
## [19] MASS_7.3-59         tibble_3.2.1        munsell_0.5.0
## [22] DBI_1.1.3           pillar_1.9.0        rlang_1.1.0
## [25] utf8_1.2.3          xfun_0.39           pkgload_1.3.2
## [28] cli_3.6.1           withr_2.5.0         magrittr_2.0.3
## [31] mgcv_1.8-42         digest_0.6.31       grid_4.3.0
## [34] lifecycle_1.0.3     nlme_3.1-162        vctrs_0.6.2
## [37] evaluate_0.20       glue_1.6.2          numDeriv_2016.8-1.1
## [40] fansi_1.0.4         colorspace_2.1-0    rmarkdown_2.21
## [43] purrr_0.3.4         tools_4.3.0         pkgconfig_2.0.3
## [46] htmltools_0.5.5
```

```
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] 2.7183
```

```
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] 0.33333
```

```
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))
```

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

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

```
## a
```

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

```
## b
```

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

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

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

```
## b
```

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

```
## a
```

```
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

```
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.

```
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(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
```

```
##
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
```

Appendix D: A comparison of nlsr::nlxb with nls and minpack::nlsLM

R has several tools for estimating nonlinear models and minimizing sums of squares functions. Sometimes we talk of **nonlinear regression** and at other times of **minimizing a sum of squares function**. Many workers conflate these two tasks. In this appendix, some of the differences between the tools available in R for these two computational tasks are highlighted. In particular, we compare the tools from the package `nlsr` (John C Nash and Duncan Murdoch (2019)), particularly function `nlxb()` with those from base-R `nls()` and the `nlsLM` function of package `minpack.lm` (Elzhov et al. (2012)). We also compare how `nlsr:nlfb()` and `minpack.lm:nls.lm` allow a sum of squares function to be minimized.

Principal differences

The main differences in the tools relate to the following features:

- the way in which derivative information is computed for the Jacobian of the modelling function
- specification of the model as an R programmatic function is unavailable in `nlsr::nlxb()`
- the use of a Marquardt stabilization for solution of the linearized least squares problem at each iteration
- details of the criterion used to terminate the iteration
- the structure of the output of the tools
- how models are predicted for new data.

Derivative information

As detailed above, `nlsr::nlxb()` attempts to use symbolic and algorithmic tools to obtain the derivatives of the model expression that are needed for the **Jacobian** matrix that is used in creating a linearized sub-problem at each iteration of an attempted solution of the minimization of the sum of squared residuals. As discussed in the section "Analytic versus approximate Jacobians" and using the code in Appendix B, `nls()` and `minpack.lm::nlsLM()` use a very simple forward-difference approximation for the partial derivatives for the Jacobian.

Forward difference approximations are less accurate than central differences, and both are subject to numerical error when the modelling function is "flat", so that there is a large amount of digit cancellation in the subtraction necessary to compute the derivative approximation.

`minpack.lm::nlsLM` uses the same derivatives as far as I can determine. The loss of information compared to the analytic or algorithmic derivatives of `nlsr::nlxb()` is important in that it can lead to Jacobian matrices that are computationally singular, where `nls()` will stop with "singular gradient". (It is actually the Jacobian which is singular here, and I will stay with that terminology.) `minpack.lm::nlsLM()` may fail to get started if the initial Jacobian is singular, but is less susceptible in general, as described in the sub-section on Marquardt stabilization which follows.

Consequences of different derivative computations While readers might expect that the precise derivative information of `nlsr::nlxb()` would mean a faster solution, this is quite often not the case. Approximate derivatives may allow faster approach to the solution by "ironing out" wrinkles in the function surface. In my opinion, the main advantage of precise derivative information is in testing that we actually have arrived at a solution.

There are even some cases where the approximation may be helpful, though users may not realize the potential danger. Thanks to Karl Schilling for an example of modelling with the function

```
a * (x ^ b)
```

where `x` is our data and we wish to estimate `a` and `b`. Now the partial derivative of this function w.r.t. `b` is

```
partialderiv <- D(expression(a * (x ^ b)),"b")
print(partialderiv)
```

```
## a * (x^b * log(x))
```

The danger here is that we may have data values `x = 0`, in which case the **derivative** is not defined, though the model can still be evaluated. Thus `nlsr::nlxb()` will not compute a solution, while `nls()` and `minpack.lm::nlsLM()` will generally proceed. A workaround is to provide a very small value instead of zero for the data, though I find this inelegant. Another approach is to drop the offending element of the data, though this risks altering the model estimated. A proper treatment might be to develop the limit of the derivative as the data value goes to zero, but finding general software that can detect and deal with this is a large project.

Timing comparisons Let us compare timings on the (scaled) Hobbs weed problem introduced in Section ??.

```
require(microbenchmark)
```

```
## Loading required package: microbenchmark
```

```
## nls on Hobbs scaled model
weed <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443,
          38.558, 50.156, 62.948, 75.995, 91.972)
tt <- 1:12
weeddf <- data.frame(y=weed, tt=tt)
wmods <- y ~ 100*b1/(1+10*b2*exp(-0.1*b3*tt))
stx<-c(b1=2, b2=5, b3=3)
tnls<-microbenchmark((anls<-nls(wmods, start=stx, data=weeddf)), unit="us")
tnls
```

```
## Unit: microseconds
##                                              expr    min     lq   mean median
##  (anls <- nls(wmods, start = stx, data = weeddf)) 833.85 843.05 863.06 847.28
##      uq    max neval
##  874.97 1073.1   100
```

```
## nlsr::nlfb() on Hobbs scaled model
tnlxb<-microbenchmark((anlxb<-nlsr::nlxb(wmods, start=stx, data=weeddf)), unit="us")
tnlxb
```

```
## Unit: microseconds
##                                                 expr    min   lq   mean
##  (anlxb <- nlsr::nlxb(wmods, start = stx, data = weeddf)) 1279.9 1293 1328.8
##  median     uq    max neval
##  1307.6 1333.7 2744.5   100
```

```
## minpack.lm::nlsLM() on Hobbs scaled model
tnlsLM<-microbenchmark((anlsLM<-minpack.lm::nlsLM(start=stx, formula=wmods, data=weeddf)),
                       unit="us")
tnlsLM
```

```
## Unit: microseconds
##                                                                      expr
##  (anlsLM <- minpack.lm::nlsLM(start = stx, formula = wmods, data = weeddf))
##     min     lq   mean median     uq    max neval
##  625.15 628.38 642.57 630.72 636.11 1064.3   100
```

Programmatic modelling functions

A consequence of the symbolic derivative approach in `nlsr::nlxb()` is that it cannot be applied to a modelling expression that includes an R function i.e., sub-program.

This limitation could be overcome if there were appropriate automatic differentiation code (to provide derivative computations based on transformation of the modelling function's programmatic form) or a mechanism to specify a form of numerical approximation. As of December 2020, it seems more likely that the latter approach will be realized first, and it is one the near-term development goals.

Functional expression of residuals and Jacobian

```
require(microbenchmark)
## nlsr::nlfb() on Hobbs scaled
# Scaled Hobbs problem
shobbs.res  <-  function(x){ # scaled Hobbs weeds problem -- residual
  # This variant uses looping
  if(length(x) != 3) stop("shobbs.res -- parameter vector n!=3")
  y  <-  c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443,
           38.558, 50.156, 62.948, 75.995, 91.972)
  tt  <-  1:12
  res  <-  100.0*x[1]/(1+x[2]*10.*exp(-0.1*x[3]*tt)) - y
}

shobbs.jac  <-  function(x) { # scaled Hobbs weeds problem -- Jacobian
  jj  <-  matrix(0.0, 12, 3)
  tt  <-  1:12
  yy  <-  exp(-0.1*x[3]*tt)
  zz  <-  100.0/(1+10.*x[2]*yy)
  jj[tt,1]   <-   zz
  jj[tt,2]   <-   -0.1*x[1]*zz*zz*yy
  jj[tt,3]   <-   0.01*x[1]*zz*zz*yy*x[2]*tt
  attr(jj, "gradient") <- jj
  jj
}

st1 <- c(b1=1, b2=1, b3=1)
tnlfb<-microbenchmark((anlfb<-nlsr::nlfb(start=st1, resfn=shobbs.res, jacfn=shobbs.jac)), unit="us")
tnlfb
```

```
## Unit: microseconds
##                                                                      expr
##  (anlfb <- nlsr::nlfb(start = st1, resfn = shobbs.res, jacfn = shobbs.jac))
##     min     lq   mean median     uq   max neval
##  4410.8 4458.5 4769.7 4509.9 4609.2 21015   100
```

```
## minpack.lm::nls.lm() on Hobbs scaled
tnls.lm<-microbenchmark((anls.lm<-minpack.lm::nls.lm(par=st1, fn=shobbs.res, jac=shobbs.jac)))
tnls.lm
```

```
## Unit: microseconds
##                                                                        expr
##  (anls.lm <- minpack.lm::nls.lm(par = st1, fn = shobbs.res, jac = shobbs.jac))
##     min     lq   mean median     uq    max neval
##  114.82 116.37 122.05 117.19 118.25 435.81   100
```

Marquardt stabilization

All three of the R functions under consideration try to minimize a sum of squares. If the model is provided in the form

`y ~ (some expression)`

then the residuals are computed by evaluating the difference between `(some expression)` and `y`. My own preference, and that of K F Gauss, is to use `(some expression) - y`. This is to avoid having to be concerned with the negative sign – the derivative of the residual defined in this way is the same as the derivative of the modelling function, and we avoid the chance of a sign error. The Jacobian matrix is made up of elements where element `i, j` is the partial derivative of residual `i` w.r.t. parameter `j`.

`nls()` attempts to minimize a sum of squared residuals by a Gauss-Newton method. If we compute a Jacobian matrix `J` and a vector of residuals `r` from a vector of parameters `x`, then we can define a linearized problem

$$J^T J\delta = -J^T r$$

This leads to an iteration where, from a set of starting parameters `x0`, we compute

$$x_{i+1} = x_i + \delta$$

This is commonly modified to use a step factor `step`

$$x_{i+1} = x_i + step * \delta$$

It is in the mechanisms to choose the size of `step` and to decide when to terminate the iteration that Gauss-Newton methods differ. Indeed, though I have tried several times, I find the very convoluted code behind `nls()` very difficult to decipher. Unfortunately, its authors have (at 2018 as far as I am aware) all ceased to maintain the code.

Both `nlsr::nlxb()` and `minpack.lm::nlsLM` use a Levenberg-Marquardt stabilization of the iteration. (Marquardt (1963), Levenberg (1944)), solving

$$(J^T J + \lambda D)\delta = -J^T r$$

where D is some diagonal matrix and lambda is a number of modest size initially. Clearly for $\lambda = 0$ we have a Gauss-Newton method. Typically, the sum of squares of the residuals calculated at the "new" set of parameters is used as a criterion for keeping those parameter values. If so, the size of λ is reduced. If not, we increase the size of λ and compute a new δ. Note that a new J, the expensive step in each iteration, is NOT required.

As for Gauss-Newton methods, the details of how to start, adjust and terminate the iteration lead to many variants, increased by different possibilities for specifying D. See Nash (1979). There are also a number of ways to solve the stabilized Gauss-Newton equations, some of which do not require the explicit $J^T J$ matrix.

Criterion used to terminate the iteration

`nls()` and `nlsr` use a form of the relative offset convergence criterion, Bates, Douglas M. and Watts, Donald G. (1981). `minpack.lm` uses a somewhat different and more complicated set of tests. Unfortunately, the relative offset criterion as implemented in `nls()` is unsuited to problems where the residuals can be zero. There are ways to work around the difficulties, and `nlsr` has used one approach. See *An illustrative nonlinear regression problem* below.

Output of the modelling functions

`nls()` and `nlsLM()` return the same solution structure. Let us examine this for one of our example results (we will choose one that does NOT have small residuals, so that all the functions "work").

```
str(nlsy0t0ax)
```

```
## List of 6
##  $ m          :List of 16
##   ..$ resid     :function ()
##   ..$ fitted    :function ()
##   ..$ formula   :function ()
##   ..$ deviance  :function ()
##   ..$ lhs       :function ()
##   ..$ gradient  :function ()
##   ..$ conv      :function ()
##   ..$ incr      :function ()
##   ..$ setVarying:function (vary = rep_len(TRUE, np))
##   ..$ setPars   :function (newPars)
##   ..$ getPars   :function ()
##   ..$ getAllPars:function ()
##   ..$ getEnv    :function ()
##   ..$ trace     :function ()
##   ..$ Rmat      :function ()
##   ..$ predict   :function (newdata = list(), qr = FALSE)
##   ..- attr(*, "class")= chr "nlsModel"
##  $ convInfo   :List of 5
##   ..$ isConv     : logi TRUE
##   ..$ finIter    : int 6
##   ..$ finTol     : num 3.9e-10
##   ..$ stopCode   : int 0
##   ..$ stopMessage: chr "converged"
##  $ data       : symbol edta
##  $ call       : language nls(formula = y1 ~ a * (t0a^b), data = edta, start = start1, control = list
##  $ dataClasses: Named chr "numeric"
##   ..- attr(*, "names")= chr "t0a"
##  $ control    :List of 7
##   ..$ maxiter    : num 10000
##   ..$ tol        : num 1e-05
##   ..$ minFactor  : num 0.000977
##   ..$ printEval  : logi FALSE
##   ..$ warnOnly   : logi FALSE
##   ..$ scaleOffset: num 0
##   ..$ nDcentral  : logi FALSE
##  - attr(*, "class")= chr "nls"
```

The `minpack.lm::nlsLM` output has the same structure, which could be revealed by the R command `str(nlsLMy1t0a)`. Note that this structure has a lot of special functions in the sub-list `m`. By contrast, the `nlsr()` output is much less flamboyant. There are, in fact, no functions as part of the structure.

```
str(nlsry1t0a)
```

```
## List of 13
##  $ resid       : num [1:20] 4.00e-05 -2.03e-09 -1.70e-09 -1.41e-09 -1.16e-09 ...
##   ..- attr(*, "gradient")= num [1:20, 1:2] 0.00001 1 1.18921 1.31607 1.41421 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "a" "b"
##  $ jacobian    : num [1:20, 1:2] 0.00001 1 1.18921 1.31607 1.41421 ...
##   ..- attr(*, "dimnames")=List of 2
```

```
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "a" "b"
##  $ feval       : num 7
##  $ jeval       : num 7
##  $ coefficients: Named num [1:2] 4 0.25
##   ..- attr(*, "names")= chr [1:2] "a" "b"
##  $ ssquares    : num 1.6e-09
##  $ lower       : num [1:2] -Inf -Inf
##  $ upper       : num [1:2] Inf Inf
##  $ maskidx     : int(0)
##  $ weights     : num [1:20] 1 1 1 1 1 1 1 1 1 1 ...
##  $ formula     :Class 'formula'  language y1 ~ a * (t0a^b)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ resfn       :function (prm)
##  $ data        : symbol edta
##  - attr(*, "class")= chr "nlsr"
```

Which of these approaches is "better" can be debated. My preference is for the results of optimization computations to be essentially data, including messages, though some tools within some of my packages will return functions for specific reasons, e.g., to return a function from an expression. However, I prefer to use specified functions such as `predict.nlsr()` below to obtain predictions. I welcome comment and discussion, as this is not, in my view, a closed topic.

Prediction

Let us predict our models at the mean of the data. Because `nlxb()` returns a different structure from that found by `nls()` and `nlsLM()` the code for `predict()` for an object from `nlsr` is different. `minpack.lm` uses `predict.nls` since the output structure of the modelling step is equivalent to that from `nls()`.

```
nudta <- colMeans(edta)
predict(nlsy0t0ax, newdata=nudta)
```

```
## [1] 7.0225
```

```
predict(nlsLMy1t0a, newdata=nudta)
```

```
## [1] 7.0225
```

```
predict(nlsry1t0a, newdata=nudta)
```

```
## [1] 7.0225
## attr(,"class")
## [1] "predict.nlsr"
## attr(,"pkgname")
## [1] "nlsr"
```

An illustrative nonlinear regression problem

So we can illustrate some of the issues, let us create some example data for a seemingly straightforward computational problem.

```
# Here we set up an example problem with data
# Define independent variable
t0 <- 0:19
t0a<-t0
t0a[1]<-1e-20 # very small value
# Drop first value in vectors
t0t<-t0[-1]
```

```
y1 <- 4 * (t0^0.25)
y1t<-y1[-1]
n <- length(t0)
fuzz <- rnorm(n)
range <- max(y1)-min(y1)
## add some "error" to the dependent variable
y1q <- y1 + 0.2*range*fuzz
edta <- data.frame(t0=t0, t0a=t0a, y1=y1, y1q=y1q)
edtat <- data.frame(t0t=t0t, y1t=y1t)
```

Let us try this example modelling `y0` against `t0`. Note that this is a zero-residual problem, so `nls()` should complain or fail, which it appears to do but by exceeding the iteration limit, which is not very communicative of the underlying issue. The `nls()` documentation warns

"Warning

Do not use nls on artificial "zero-residual" data."

It goes on to recommend that users add "error" to the data to avoid such problems. I feel this is a very unsatisfactory kludge. It is NOT due to a genuine mathematical issue, but due to the relative offset convergence criterion used to terminate the method. In October 2020, I suggested a patch for nls() to R-core that it seems will become part of base R eventually. This patch allows the user to specify a parameter, tentatively named `convTestAdd` with a zero default value, in `nls.control()`. (This allows existing examples using `nsl()` to function without change.) A small positive value for this control parameter avoids a zero divided by zero issue in the relative offset convergence test in `nls()` used to terminate iterations. This adjustment to the convergence test has been in `nlsr` since its creation.

Here is the output.

nls

```
cprint <- function(obj){
   # print object if it exists
  sobj<-deparse(substitute(obj))
  if (exists(sobj)) {
      print(obj)
  } else {
      cat(sobj," does not exist\n")
  }
#  return(NULL)
}
start1 <- c(a=1, b=1)
try(nlsy0t0 <- nls(formula=y1~a*(t0^b), start=start1, data=edta))
```

```
## Error in nls(formula = y1 ~ a * (t0^b), start = start1, data = edta) :
##   number of iterations exceeded maximum of 50
```

```
cprint(nlsy0t0)
```

```
## nlsy0t0  does not exist
```

```
# Since this fails to converge, let us increase the maximum iterations
try(nlsy0t0x <- nls(formula=y1~a*(t0^b), start=start1, data=edta,
                    control=nls.control(maxiter=10000)))
```

```
## Error in nls(formula = y1 ~ a * (t0^b), start = start1, data = edta, control = nls.control(maxiter =
##   number of iterations exceeded maximum of 10000
```

```
cprint(nlsy0t0x)
```

```
## nlsy0t0x  does not exist
```

```
try(nlsy0t0ax <- nls(formula=y1~a*(t0a^b), start=start1, data=edta,
                     control=nls.control(maxiter=10000)))
cprint(nlsy0t0ax)
```

```
## Nonlinear regression model
##   model: y1 ~ a * (t0a^b)
##    data: edta
##    a    b
## 4.00 0.25
##  residual sum-of-squares: 1.6e-09
##
## Number of iterations to convergence: 6
## Achieved convergence tolerance: 3.9e-10
```

```
try(nlsy0t0t <- nls(formula=y1t~a*(t0t^b), start=start1, data=edtat))
```

```
## Error in nls(formula = y1t ~ a * (t0t^b), start = start1, data = edtat) :
##   number of iterations exceeded maximum of 50
```

```
cprint(nlsy0t0t)
```

```
## nlsy0t0t  does not exist
```

nlsr

```
nlsry1t0 <- try(nlxb(formula=y1~a*(t0^b), start=start1, data=edta))
```

```
## Error in model2rjfun(formula, pnum, data = data) : Jacobian contains NaN
```

```
cprint(nlsry1t0)
```

```
## [1] "Error in model2rjfun(formula, pnum, data = data) : Jacobian contains NaN\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in model2rjfun(formula, pnum, data = data): Jacobian contains NaN>
```

```
nlsry1t0a <- nlxb(formula=y1~a*(t0a^b), start=start1, data=edta)
cprint(nlsry1t0a)
```

```
## residual sumsquares =  1.6e-09  on  20 observations
##     after  7    Jacobian and  7 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a                     4     4.811e-06     831443   1.02e-96  -2.391e-13       72.97
## b                  0.25     5.011e-07     498907  1.003e-92  -9.371e-13        1.95
```

```
nlsry1t0t <- nlxb(formula=y1t~a*(t0t^b), start=start1, data=edtat)
cprint(nlsry1t0t)
```

```
## residual sumsquares =  6.3766e-27  on  19 observations
##     after  7    Jacobian and  7 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a                     4     9.883e-15  4.048e+14  2.612e-239  -2.416e-13       72.97
## b                  0.25     1.029e-15  2.429e+14  1.541e-235  -8.636e-13        1.95
```

minpack.lm

```
library(minpack.lm)
nlsLMy1t0 <- nlsLM(formula=y1~a*(t0^b), start=start1, data=edta)
nlsLMy1t0
```

```
## Nonlinear regression model
##   model: y1 ~ a * (t0^b)
##    data: edta
##    a    b
## 4.00 0.25
##  residual sum-of-squares: 0
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 1.49e-08
```

```
nlsLMy1t0a <- nlsLM(formula=y1~a*(t0a^b), start=start1, data=edta)
nlsLMy1t0a
```

```
## Nonlinear regression model
##   model: y1 ~ a * (t0a^b)
##    data: edta
##    a    b
## 4.00 0.25
##  residual sum-of-squares: 1.6e-09
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 1.49e-08
```

```
nlsLMy1t0t <- nlsLM(formula=y1t~a*(t0t^b), start=start1, data=edtat)
nlsLMy1t0t
```

```
## Nonlinear regression model
##   model: y1t ~ a * (t0t^b)
##    data: edtat
##    a    b
## 4.00 0.25
##  residual sum-of-squares: 0
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 1.49e-08
```

We have seemingly found a workaround for our difficulty, but I caution that initially I found very unsatisfactory results when I set the "very small value" to 1.0e-7. The correct approach is clearly to understand what is going on. Getting computers to provide that understanding is a serious challenge.

Problems that are NOT regressions

Some nonlinear least squares problems are NOT nonlinear regressions. That is, we do not have a formula `y ~ (some function)` to define the problem. This is another reason to use the residual in the form `(some function) - y` In many cases of interest we have no `y`.

The Brown and Dennis test problem (Moré, Garbow, and Hillstrom (1981), problem 16) is of this form. Suppose we have `m` observations, then we create a scaled index `t` which is the "data" for the function. To run the nonlinear least squares functions that use a formula, we do, however, need a "y" variable. Clearly adding zero to the residual will not change the problem, so we set the data for "y" as all zeros. Note that `nls()` and `nlsLM()` need some extra iterations to find the solution to this somewhat nasty problem.

```
m <- 20
t <- seq(1, m) / 5
y <- rep(0,m)
library(nlsr)
library(minpack.lm)

bddata <- data.frame(t=t, y=y)
bdform <- y ~ ((x1 + t * x2 - exp(t))^2 + (x3 + x4 * sin(t) - cos(t))^2)
prm0 <- c(x1=25, x2=5, x3=-5, x4=-1)
fbd <-model2ssgrfun(bdform, prm0, bddata)
cat("initial sumsquares=",as.numeric(crossprod(fbd(prm0))),"\n")
```

```
## initial sumsquares= 6.2832e+13
```

```
nlsrbd <- nlxb(bdform, start=prm0, data=bddata, trace=FALSE)
nlsrbd
```

```
## residual sumsquares =  85822  on  20 observations
##     after  28    Jacobian and  46 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## x1              -11.5944         4.017     -2.886    0.01075   0.0005499         176
## x2               13.2036         1.231      10.73  1.025e-08  -0.0002686        28.1
## x3             -0.403439         28.08   -0.01437     0.9887    0.001689       3.917
## x4              0.236779         39.79   0.005951     0.9953    0.000661       1.624
```

```
nlsbd10k <- nls(bdform, start=prm0, data=bddata, trace=FALSE,
                control=nls.control(maxiter=10000))
nlsbd10k
```

```
## Nonlinear regression model
##   model: y ~ ((x1 + t * x2 - exp(t))^2 + (x3 + x4 * sin(t) - cos(t))^2)
##    data: bddata
##      x1      x2      x3      x4
## -11.594  13.204  -0.403   0.237
##  residual sum-of-squares: 85822
##
## Number of iterations to convergence: 867
## Achieved convergence tolerance: 9.76e-06
```

```
nlsLMbd10k <- nlsLM(bdform, start=prm0, data=bddata, trace=FALSE,
                    control=nls.lm.control(maxiter=10000, maxfev=10000))
```

```
## Warning in nls.lm(par = start, fn = FCT, jac = jac, control = control, lower =
## lower, : resetting `maxiter' to 1024!
```

```
nlsLMbd10k
```

```
## Nonlinear regression model
##   model: y ~ ((x1 + t * x2 - exp(t))^2 + (x3 + x4 * sin(t) - cos(t))^2)
##    data: bddata
##      x1      x2      x3      x4
## -11.592  13.203  -0.404   0.237
##  residual sum-of-squares: 85822
##
## Number of iterations to convergence: 242
## Achieved convergence tolerance: 1.49e-08
```

Let us try predicting the "residual" for some new data.

```
ndata <- data.frame(t=c(5,6), y=c(0,0))
predict(nlsLMbd10k, newdata=ndata)
```

```
## [1]   8835.3 112766.9
```

```
# now nls
predict(nlsbd10k, newdata=ndata)
```

```
## [1]   8834.9 112764.7
```

```
# now nlsr
predict(nlsrbd, newdata=ndata)
```

```
## [1]   8834.9 112764.7
## attr(,"class")
## [1] "predict.nlsr"
## attr(,"pkgname")
## [1] "nlsr"
```

We could, of course, try setting up a different formula, since the "residuals" can be computed in any way such that their absolute value is the same. Therefore we could try moving the exponential part of the function for each equation to the left hand side as in `bdf2` below.

```
bdf2 <-  (x1 + t * x2 - exp(t))^2 ~ - (x3 + x4 * sin(t) - cos(t))^2
```

However, we discover that the parsing of the model formula fails for this formulation.

A check on the Brown and Dennis calculation via function minimization

We can attack the Brown and Dennis problem by applying nonlinear function minimization programs to the sum of squared "residuals" as a function of the parameters. The code below does this. We omit the output for space reasons.

```
#' Brown and Dennis Function
#'
#' Test function 16 from the More', Garbow and Hillstrom paper.
#'
#' The objective function is the sum of \code{m} functions, each of \code{n}
#' parameters.
#'
#' \itemize{
#'   \item Dimensions: Number of parameters \code{n = 4}, number of summand
#'   functions \code{m >= n}.
#'   \item Minima: \code{f = 85822.2} if \code{m = 20}.
#' }
#'
#' @param m Number of summand functions in the objective function. Should be
#'   equal to or greater than 4.
#' @return A list containing:
#' \itemize{
#'   \item \code{fn} Objective function which calculates the value given input
#'   parameter vector.
#'   \item \code{gr} Gradient function which calculates the gradient vector
#'   given input parameter vector.
#'   \item \code{fg} A function which, given the parameter vector, calculates
#'   both the objective value and gradient, returning a list with members
#'   \code{fn} and \code{gr}, respectively.
```

```
#'   \item \code{x0} Standard starting point.
#' }
#' @references
#' More', J. J., Garbow, B. S., & Hillstrom, K. E. (1981).
#' Testing unconstrained optimization software.
#' \emph{ACM Transactions on Mathematical Software (TOMS)}, \emph{7}(1), 17-41.
#' \url{https://doi.org/10.1145/355934.355936}
#'
#' Brown, K. M., & Dennis, J. E. (1971).
#' \emph{New computational algorithms for minimizing a sum of squares of
#' nonlinear functions} (Report No. 71-6).
#' New Haven, CT: Department of Computer Science, Yale University.
#'
#' @examples
#' # Use 10 summand functions
#' fun <- brown_den(m = 10)
#' # Optimize using the standard starting point
#' x0 <- fun$x0
#' res_x0 <- stats::optim(par = x0, fn = fun$fn, gr = fun$gr, method =
#' "L-BFGS-B")
#' # Use your own starting point
#' res <- stats::optim(c(0.1, 0.2, 0.3, 0.4), fun$fn, fun$gr, method =
#' "L-BFGS-B")
#'
#' # Use 20 summand functions
#' fun20 <- brown_den(m = 20)
#' res <- stats::optim(fun20$x0, fun20$fn, fun20$gr, method = "L-BFGS-B")
#' @export
#`
brown_den <- function(m = 20) {
  list(
    fn = function(par) {
      x1 <- par[1]
      x2 <- par[2]
      x3 <- par[3]
      x4 <- par[4]

      ti <- (1:m) * 0.2
      l <- x1 + ti * x2 - exp(ti)
      r <- x3 + x4 * sin(ti) - cos(ti)
      f <- l * l + r * r
      sum(f * f)
    },
    gr = function(par) {
      x1 <- par[1]
      x2 <- par[2]
      x3 <- par[3]
      x4 <- par[4]

      ti <- (1:m) * 0.2
      sinti <- sin(ti)
      l <- x1 + ti * x2 - exp(ti)
      r <- x3 + x4 * sinti - cos(ti)
```

```
      f <- l * l + r * r
      lf4 <- 4 * l * f
      rf4 <- 4 * r * f
      c(
        sum(lf4),
        sum(lf4 * ti),
        sum(rf4),
        sum(rf4 * sinti)
      )
    },
    fg = function(par) {
      x1 <- par[1]
      x2 <- par[2]
      x3 <- par[3]
      x4 <- par[4]

      ti <- (1:m) * 0.2
      sinti <- sin(ti)
      l <- x1 + ti * x2 - exp(ti)
      r <- x3 + x4 * sinti - cos(ti)
      f <- l * l + r * r
      lf4 <- 4 * l * f
      rf4 <- 4 * r * f

      fsum <- sum(f * f)
      grad <- c(
        sum(lf4),
        sum(lf4 * ti),
        sum(rf4),
        sum(rf4 * sinti)
      )

      list(
        fn = fsum,
        gr = grad
      )
    },
    x0 = c(25, 5, -5, 1)
  )
}
mbd <- brown_den(m=20)
mbd
mbd$fg(mbd$x0)
bdsolnm <- optim(mbd$x0, mbd$fn, control=list(trace=0))
bdsolnm
bdsolbfgs <- optim(mbd$x0, mbd$fn, method="BFGS", control=list(trace=0))
bdsolbfgs

library(optimx)
methlist <- c("Nelder-Mead","BFGS","Rvmmin","L-BFGS-B","Rcgmin","ucminf")

solo <- opm(mbd$x0, mbd$fn, mbd$gr, method=methlist, control=list(trace=0))
summary(solo, order=value)
```

```
## A failure above is generally because a package in the 'methlist' is not installed.
```

References

Bates, Douglas M., and Watts, Donald G. 1981. "A Relative Offset Orthogonality Convergence Criterion for Nonlinear Least Squares." *Technometrics* 23 (2): 179–83.

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

Elzhov, Timur V., Katharine M. Mullen, Andrej-Nikolai Spiess, and Ben Bolker. 2012. *Minpack.lm: R Interface to the Levenberg-Marquardt Nonlinear Least-Squares Algorithm Found in Minpack, Plus Support for Bounds.* R Project for Statistical Computing. http://CRAN.R-project.org/package=minpack.lm.

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.

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

Levenberg, Kenneth. 1944. "A Method for the Solution of Certain Non-Linear Problems in Least Squares." *Quarterly of Applied Mathematics* 2: 164–68.

Marquardt, Donald W. 1963. "An Algorithm for Least-Squares Estimation of Nonlinear Parameters." *SIAM Journal on Applied Mathematics* 11 (2): 431–41.

Moré, Jorge J., Burton S. Garbow, and Kenneth E. Hillstrom. 1981. "Testing Unconstrained Optimization Software." *J-Toms* 7 (1): 17–41.

Nash, J. C. 1979. *Compact Numerical Methods for Computers : Linear Algebra and Function Minimisation.* Book. Bristol: Hilger: Bristol.

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.