nlsr Background, Development, Examples and Discussion

John C. Nash

2021-11-22

This rather long vignette is to explain the development and testing of **nlsr**, an R package to try to bring the **R** function nls() up to date and to add capabilities for the extension of the symbolic and automatic derivative tools in **R**.

A particular goal in **nlsr** is to attempt, wherever possible, to use analytic or automatic derivatives. The function nls() uses a rather weak forward derivative approximation. A second objective is to use a Marquardt stabilization of the Gauss-Newton equations to avoid the commonly encountered "singular gradient" failure of nls(). This refers to the loss of rank of the Jacobian at the parameters for evaluation. The particular stabilization also incorporates a very simple trick to avoid very small diagonal elements of the Jacobian inner product, though in the present implementations, this is accomplished indirectly. See the section below **Implementation of method**

In preparing the **nlsr** package there is a sub-goal to unify, or at least render compatible, various packages in **R** for the estimation or analysis of problems amenable to nonlinear least squares solution.

A large part of the work for this package – particularly the parts concerning derivatives and R language structures – was initially carried out by Duncan Murdoch. Without his input, much of the capability of the package would not exist.

The package and this vignette are a work in progress, and assistance and examples are welcome. Note that there is similar work in the package Deriv (Andrew Clausen and Serguei Sokol (2015) Symbolic Differentiation, version 2.0, https://cran.r-project.org/package=Deriv), and making the current work "play nicely" with that package is desirable.

As a mechanism for highlighting issues that remain to be resolved, I have put double question marks (??) where I believe attention is needed in this document.

TODOS

- explain JN approach to relative offset convergence criterion
- how to insert numerical derivatives when Deriv unable to get result
- approximations for jacfn beyond fwd approximation. How to specify??
- how to force numerical approximations in nlfb() in a manner consistent with that used in `optimx::optimr()`, that is, to surround the name of `jacfn` with quotes if it is a numerical approximation, or to provide a logical control to `nlxb()` for this purpose.

Summary of capabilities in `nlsr`

Functions in nlsr

coef.nlsr: extracts and displays the coefficients for a model estimated by nlxb() or nlfb() in the nlsr structured object.

```
library(nlsr)
ydat<-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)
```

```
tdat<-1:12
# try setting t and y here
t <- tdat
y <- ydat
sol1 <- nlxb(y~100*b1/(1+10*b2*exp(-0.1*b3*t)), start=c(b1=2, b2=5, b3=3))
coef(sol1)
```

```
##     b1     b2     b3
## 1.9619 4.9092 3.1357
## attr(,"pkgname")
## [1] "nlsr"
```

```
print(coef(sol1))
```

```
##     b1     b2     b3
## 1.9619 4.9092 3.1357
## attr(,"pkgname")
## [1] "nlsr"
```

nlsDeriv, codeDeriv, fnDeriv, newDeriv: Functions **Deriv** and **fnDeriv** are designed as replacements for the stats package functions **D** and **deriv** respectively, though the argument lists do not match exactly. **newDeriv** allows additional analytic derivative expressions to be added. The following is an expanded and commented version of the examples from the manual for these functions.

```
require(nlsr)
newDeriv() # a call with no arguments returns a list of available functions
```

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

```
           # for which derivatives are currently defined
newDeriv(sin(x)) # a call with a function that is in the list of available derivatives
```

```
## $expr
## sin(x)
##
## $argnames
## [1] "x"
##
## $required
## [1] 1
##
## $deriv
## cos(x) * D(x)
```

```
                # returns the derivative expression for that function
nlsDeriv(~ sin(x+y), "x") # partial derivative of this function w.r.t. "x"
```

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

```
## CAUTION !! ##
newDeriv(joe(x)) # but an undefined function returns NULL
```

```
## NULL
```

```
newDeriv(joe(x), deriv=log(x^2)) # We can define derivatives, even if joe(x) is meanin
nlsDeriv(~ joe(x+z), "x")
```

```
## log((x + z)^2)
```

```
# Some examples of usage
f <- function(x) x^2
newDeriv(f(x), 2*x*D(x))
nlsDeriv(~ f(abs(x)), "x")
```

```
## 2 * abs(x) * sign(x)
```

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

```
## 2 * x
```

```
nlsDeriv(~ (abs(x)^2), "x")
```

```
## 2 * abs(x) * sign(x)
```

```
# derivatives of distribution functions
nlsDeriv(~ pnorm(x, sd=2, log = TRUE), "x")
```

```
## 0.5 * exp(dnorm(x/2, log = TRUE) - pnorm(x/2, log = TRUE))
```

```
# get evaluation code from a formula
codeDeriv(~ pnorm(x, sd = sd, log = TRUE), "x")
```

```
## {
##     .expr1 <- x/sd
##     .value <- pnorm(x, sd = sd, log = TRUE)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 1/sd * exp(dnorm(.expr1, log = TRUE) - pnorm(.expr1,
##         log = TRUE))
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
# wrap it in a function call
fnDeriv(~ pnorm(x, sd = sd, log = TRUE), "x")
```

```
## function (x, sd)
## {
##     .expr1 <- x/sd
##     .value <- pnorm(x, sd = sd, log = TRUE)
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 1/sd * exp(dnorm(.expr1, log = TRUE) - pnorm(.expr1,
##         log = TRUE))
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
f <- fnDeriv(~ pnorm(x, sd = sd, log = TRUE), "x", args = alist(x =, sd = 2))
f
```

```
## function (x, sd = 2)
## {
##     .expr1 <- x/sd
##     .value <- pnorm(x, sd = sd, log = TRUE)
```

```
##     .grad <- array(0, c(length(.value), 1L), list(NULL, "x"))
##     .grad[, "x"] <- 1/sd * exp(dnorm(.expr1, log = TRUE) - pnorm(.expr1,
##         log = TRUE))
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
f(1)
```

```
## [1] -0.36895
## attr(,"gradient")
##            x
## [1,] 0.25458
```

```
100*(f(1.01) - f(1))  # Should be close to the gradient
```

```
## [1] 0.25394
## attr(,"gradient")
##           x
## [1,] 0.2533
```

```
                      # The attached gradient attribute (from f(1.01)) is
                      # meaningless after the subtraction.
```

model2rjfun, model2ssgrfun, modelexpr, rjfundoc These functions create functions to evaluate residuals or sums of squares at particular parameter locations.

```
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)
ii <- 1:12
wdf <- data.frame(weed, ii)
wmod <- weed ~ b1/(1+b2*exp(-b3*ii))
start1  <-  c(b1=1, b2=1, b3=1)
wrj <- model2rjfun(wmod, start1, data=wdf)
print(wrj)
```

```
## function (prm)
## {
##     if (is.null(names(prm)))
##         names(prm) <- names(pvec)
##     localdata <- list2env(as.list(prm), parent = data)
##     eval(residexpr, envir = localdata)
## }
## <bytecode: 0x55c43b156848>
## <environment: 0x55c43dc42b88>
```

```
weedux <- nlxb(wmod, start=c(b1=200, b2=50, b3=0.3))
print(weedux)
```

```
## nlsr object: x
## residual sumsquares =  2.5873  on  12 observations
##     after  6    Jacobian and  7 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               196.186         11.31       17.35  3.167e-08  -1.334e-10        1011
## b2               49.0916         1.688       29.08  3.284e-10  -3.589e-09      0.4605
## b3               0.31357      0.006863       45.69  5.768e-12    4.09e-07     0.04714
```

```
wss <- model2ssgrfun(wmod, start1, data=wdf)
print(wss)
```

```
## function (prm)
## {
##     resids <- rjfun(prm)
##     ss <- as.numeric(crossprod(resids))
##     if (gradient) {
##         jacval <- attr(resids, "gradient")
##         grval <- 2 * as.numeric(crossprod(jacval, resids))
##         attr(ss, "gradient") <- grval
##     }
##     attr(ss, "resids") <- resids
##     ss
## }
## <bytecode: 0x55c43cbd5470>
## <environment: 0x55c43cbd3fe0>
```

```
# We can get expressions used to calculate these as follows:
wexpr.rj <- modelexpr(wrj)
print(wexpr.rj)
```

```
## expression({
##     .expr3 <- exp(-b3 * ii)
##     .expr5 <- 1 + b2 * .expr3
##     .expr10 <- .expr5^2
##     .value <- b1/.expr5 - weed
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("b1",
##         "b2", "b3")))
##     .grad[, "b1"] <- 1/.expr5
##     .grad[, "b2"] <- -(b1 * .expr3/.expr10)
##     .grad[, "b3"] <- b1 * (b2 * (.expr3 * ii))/.expr10
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
wexpr.ss <- modelexpr(wss)
print(wexpr.ss)
```

```
## expression({
##     .expr3 <- exp(-b3 * ii)
##     .expr5 <- 1 + b2 * .expr3
##     .expr10 <- .expr5^2
##     .value <- b1/.expr5 - weed
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("b1",
##         "b2", "b3")))
##     .grad[, "b1"] <- 1/.expr5
##     .grad[, "b2"] <- -(b1 * .expr3/.expr10)
##     .grad[, "b3"] <- b1 * (b2 * (.expr3 * ii))/.expr10
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

```
wrjdoc <- rjfundoc(wrj)
print(wrjdoc)
```

```
## FUNCTION wrj
## Formula: weed ~ b1/(1 + b2 * exp(-b3 * ii))
## Code:        expression({
##     .expr3 <- exp(-b3 * ii)
##     .expr5 <- 1 + b2 * .expr3
##     .expr10 <- .expr5^2
##     .value <- b1/.expr5 - weed
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("b1",
##         "b2", "b3")))
##     .grad[, "b1"] <- 1/.expr5
##     .grad[, "b2"] <- -(b1 * .expr3/.expr10)
##     .grad[, "b3"] <- b1 * (b2 * (.expr3 * ii))/.expr10
##     attr(.value, "gradient") <- .grad
##     .value
## })
## Parameters:   b1, b2, b3
## Data:         weed, ii
##
## VALUES
## Observations:     12
## Parameters:
## b1 b2 b3
##  1  1  1
## Data (length 12):
##      weed ii
## 1   5.308  1
## 2   7.240  2
## 3   9.638  3
## 4  12.866  4
## 5  17.069  5
## 6  23.192  6
## 7  31.443  7
## 8  38.558  8
## 9  50.156  9
## 10 62.948 10
## 11 75.995 11
## 12 91.972 12
## FUNCTION wrj
## Formula: weed ~ b1/(1 + b2 * exp(-b3 * ii))
## Code:        expression({
##     .expr3 <- exp(-b3 * ii)
##     .expr5 <- 1 + b2 * .expr3
##     .expr10 <- .expr5^2
##     .value <- b1/.expr5 - weed
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("b1",
##         "b2", "b3")))
##     .grad[, "b1"] <- 1/.expr5
##     .grad[, "b2"] <- -(b1 * .expr3/.expr10)
##     .grad[, "b3"] <- b1 * (b2 * (.expr3 * ii))/.expr10
##     attr(.value, "gradient") <- .grad
##     .value
## })
## Parameters:   b1, b2, b3
```

```
## Data:         weed, ii
##
## VALUES
## Observations:     12
## Parameters:
## b1 b2 b3
##  1  1  1
## Data (length 12):
##      weed ii
## 1   5.308  1
## 2   7.240  2
## 3   9.638  3
## 4  12.866  4
## 5  17.069  5
## 6  23.192  6
## 7  31.443  7
## 8  38.558  8
## 9  50.156  9
## 10 62.948 10
## 11 75.995 11
## 12 91.972 12
```

```
# We do not have similar function for ss functions
```

nlfb Given a nonlinear model expressed as an expression of the form of a function that computes the residuals from the model and a start vector `par` , tries to minimize the nonlinear sum of squares of these residuals w.r.t. `par`. If `model(par, mydata)` computes an a vector of numbers that are presumed to be able to fit data `lhs`, then the residual vector is `(model(par,mydata) - lhs)` , though traditionally we write the negative of this vector. (Writing it this way allows the derivatives of the residuals w.r.t. the parameters `par` to be the same as those for `model(par,mydata).`)nlfb‘ tries to minimize the sum of squares of the residuals with respect to the parameters.

The method takes a parameter `jacfn` which returns the Jacobian matrix of derivatives of the residuals w.r.t. the parameters in an attribute `gradient`. If this is NULL, then a numerical approximation to derivatives is used. (??which – and can we use the character method to define these?? What about for nlxb??) Putting the Jacobian in the attribute `gradient` allows us to combine the computation of the residual and Jacobian in the same code block if we wish, since there are generally common computations, though it does make the setup slightly more complicated. See the example below.

The start vector preferably uses named parameters (especially if there is an underlying formula). The attempted minimization of the sum of squares uses the Nash variant (Nash, 1979) of the Marquardt algorithm, where the linear sub-problem is solved by a qr method. We explain how this is done later, as well as giving a short discussion of the relative offset convergence criterion.

```
shobbs.res  <-  function(x){ # scaled Hobbs weeds problem -- residual
  # This variant uses looping
  if(length(x) != 3) stop("hobbs.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
}

cat("try nlfb\n")
```

```
## try nlfb
```

```
st  <-  c(b1=1, b2=1, b3=1)
low  <-  -Inf
up <- Inf
ans1 <- nlfb(st, shobbs.res, shobbs.jac, trace=TRUE)
```

```
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Start:lamda: 1e-04  SS= 10685  at  b1 = 1  b2 = 1  b3 = 1  1 / 0
## lamda: 0.001  SS= 177800  at  b1 = 7.1383  b2 = 9.1157  b3 = 3.7692  2 / 1
## lamda: 0.01  SS= 19087  at  b1 = 1.4903  b2 = 3.072  b3 = 4.9249  3 / 1
## <<lamda: 0.004  SS= 1273.6  at  b1 = 0.79487  b2 = 2.0743  b3 = 4.761  4 / 1
## lamda: 0.04  SS= 4265  at  b1 = 1.0624  b2 = 1.9667  b3 = 2.3468  5 / 2
## <<lamda: 0.016  SS= 946.57  at  b1 = 0.96631  b2 = 2.7886  b3 = 3.5088  6 / 2
## <<lamda: 0.0064  SS= 43.28  at  b1 = 1.3414  b2 = 4.0037  b3 = 3.5588  7 / 3
## <<lamda: 0.00256  SS= 17.386  at  b1 = 1.5631  b2 = 4.485  b3 = 3.3895  8 / 4
## <<lamda: 0.001024  SS= 6.6368  at  b1 = 1.759  b2 = 4.6938  b3 = 3.2472  9 / 5
## <<lamda: 0.0004096  SS= 3.0719  at  b1 = 1.8946  b2 = 4.8321  b3 = 3.1681  10 / 6
## <<lamda: 0.00016384  SS= 2.5977  at  b1 = 1.9503  b2 = 4.8957  b3 = 3.1413  11 / 7
## <<lamda: 6.5536e-05  SS= 2.5873  at  b1 = 1.961  b2 = 4.9082  b3 = 3.1362  12 / 8
## <<lamda: 2.6214e-05  SS= 2.5873  at  b1 = 1.9618  b2 = 4.9091  b3 = 3.1357  13 / 9
## <<lamda: 1.0486e-05  SS= 2.5873  at  b1 = 1.9619  b2 = 4.9092  b3 = 3.1357  14 / 10
## <<lamda: 4.1943e-06  SS= 2.5873  at  b1 = 1.9619  b2 = 4.9092  b3 = 3.1357  15 / 11
## lamda: 4.1943e-05  SS= 2.5873  at  b1 = 1.9619  b2 = 4.9092  b3 = 3.1357  16 / 12
```

```
summary(ans1)
```

```
## $residuals
##  [1]  0.011900 -0.032755  0.092030  0.208782  0.392634 -0.057594 -1.105728
##  [8]  0.715786 -0.107648 -0.348396  0.652593 -0.287568
##
## $sigma
## [1] 0.53617
##
## $df
## [1] 3 9
##
## $cov.unscaled
##           b1        b2        b3
## b1  0.044472  0.047835 -0.025281
## b2  0.047835  0.099167 -0.017629
```

```
## b3 -0.025281 -0.017629  0.016386
##
## $param
##    Estimate Std. Error t value   Pr(>|t|)
## b1   1.9619   0.113069  17.351 3.1667e-08
## b2   4.9092   0.168844  29.075 3.2836e-10
## b3   3.1357   0.068633  45.688 5.7676e-12
##
## $resname
## [1] "ans1"
##
## $ssquares
## [1] 2.5873
##
## $nobs
## [1] 12
##
## $ct
## [1] " " " " " " "
##
## $mt
## [1] " " " " " " "
##
## $Sd
## [1] 130.1160   6.1654   2.7353
##
## $gr
##             [,1]
## [1,] -3.9578e-08
## [2,] -6.9301e-08
## [3,]  8.8404e-08
##
## $jeval
## [1] 12
##
## $feval
## [1] 16
##
## attr(,"pkgname")
## [1] "nlsr"
## attr(,"class")
## [1] "summary.nlsr"
```

```
## No jacobian function -- use internal approximation
ans1n <- nlfb(st, shobbs.res, trace=TRUE, control=list(watch=FALSE)) # NO jacfn
```

```
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Using default jacobian approximation
## Start:lamda: 1e-04  SS= 10685  at  b1 = 1  b2 = 1  b3 = 1  1 / 0
## lamda: 0.001  SS= 177800  at  b1 = 7.1383  b2 = 9.1157  b3 = 3.7692  2 / 1
## lamda: 0.01  SS= 19087  at  b1 = 1.4903  b2 = 3.072  b3 = 4.9249  3 / 1
## <<lamda: 0.004  SS= 1273.6  at  b1 = 0.79487  b2 = 2.0743  b3 = 4.761  4 / 1
## lamda: 0.04  SS= 4265  at  b1 = 1.0624  b2 = 1.9667  b3 = 2.3468  5 / 2
```

```
## <<lamda: 0.016  SS= 946.57  at  b1 = 0.96631  b2 = 2.7886  b3 = 3.5088  6 / 2
## <<lamda: 0.0064  SS= 43.28  at  b1 = 1.3414  b2 = 4.0037  b3 = 3.5588  7 / 3
## <<lamda: 0.00256  SS= 17.386  at  b1 = 1.5631  b2 = 4.485  b3 = 3.3895  8 / 4
## <<lamda: 0.001024  SS= 6.6368  at  b1 = 1.759  b2 = 4.6938  b3 = 3.2472  9 / 5
## <<lamda: 0.0004096  SS= 3.0719  at  b1 = 1.8946  b2 = 4.8321  b3 = 3.1681  10 / 6
## <<lamda: 0.00016384  SS= 2.5977  at  b1 = 1.9503  b2 = 4.8957  b3 = 3.1413  11 / 7
## <<lamda: 6.5536e-05  SS= 2.5873  at  b1 = 1.961  b2 = 4.9082  b3 = 3.1362  12 / 8
## <<lamda: 2.6214e-05  SS= 2.5873  at  b1 = 1.9618  b2 = 4.9091  b3 = 3.1357  13 / 9
## <<lamda: 1.0486e-05  SS= 2.5873  at  b1 = 1.9619  b2 = 4.9092  b3 = 3.1357  14 / 10
## <<lamda: 4.1943e-06  SS= 2.5873  at  b1 = 1.9619  b2 = 4.9092  b3 = 3.1357  15 / 11
## <<lamda: 1.6777e-06  SS= 2.5873  at  b1 = 1.9619  b2 = 4.9092  b3 = 3.1357  16 / 12
```

```
summary(ans1n)
```

```
## $residuals
##  [1]  0.011900 -0.032755  0.092030  0.208782  0.392634 -0.057594 -1.105728
##  [8]  0.715786 -0.107648 -0.348396  0.652593 -0.287568
##
## $sigma
## [1] 0.53617
##
## $df
## [1] 3 9
##
## $cov.unscaled
##           b1        b2        b3
## b1  0.044472  0.047835 -0.025281
## b2  0.047835  0.099167 -0.017629
## b3 -0.025281 -0.017629  0.016386
##
## $param
##    Estimate Std. Error t value   Pr(>|t|)
## b1   1.9619   0.113069  17.351 3.1667e-08
## b2   4.9092   0.168844  29.075 3.2836e-10
## b3   3.1357   0.068633  45.688 5.7676e-12
##
## $resname
## [1] "ans1n"
##
## $ssquares
## [1] 2.5873
##
## $nobs
## [1] 12
##
## $ct
## [1] " " " " " " " "
##
## $mt
## [1] " " " " " " " "
##
## $Sd
## [1] 130.1160   6.1654   2.7353
##
## $gr
```

```
##             [,1]
## [1,] -1.9114e-10
## [2,] -1.8499e-11
## [3,]  1.6368e-10
##
## $jeval
## [1] 12
##
## $feval
## [1] 16
##
## attr(,"pkgname")
## [1] "nlsr"
## attr(,"class")
## [1] "summary.nlsr"
```

```
## difference
coef(ans1)-coef(ans1n)
```

```
##          b1          b2          b3
## -6.1516e-11 -5.5360e-10 -6.1774e-11
## attr(,"pkgname")
## [1] "nlsr"
```

nlxb Given a nonlinear model expressed as an expression of the form lhs ~ formula_for_rhs and a start vector where parameters used in the model formula are named, attempts to find the minimum of the residual sum of squares using the Nash variant (Nash, 1979) of the Marquardt algorithm, where the linear sub-problem is solved by a qr method. The process is to develop the residual and Jacobian functions using `model2rjfun`, then call `nlfb`.

```
# Data for Hobbs problem
ydat  <-  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) # for testing
tdat  <-  seq_along(ydat) # for testing
start1  <-  c(b1=1, b2=1, b3=1)
eunsc  <-   y ~ b1/(1+b2*exp(-b3*tt))
weeddata1  <-  data.frame(y=ydat, tt=tdat)
anlxb1  <-  try(nlxb(eunsc, start=start1, trace=TRUE, data=weeddata1))
```

```
## formula: y ~ b1/(1 + b2 * exp(-b3 * tt))
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## $watch
## [1] FALSE
##
## $phi
## [1] 1
##
## $lamda
## [1] 1e-04
##
## $offset
## [1] 100
##
## $laminc
## [1] 10
```

```
##
## $lamdec
## [1] 4
##
## $femax
## [1] 10000
##
## $jemax
## [1] 5000
##
## $rofftest
## [1] TRUE
##
## $smallsstest
## [1] TRUE
##
## Finished masks check
## datvar:[1] "y"  "tt"
## Data variable  y : [1]  5.308  7.240  9.638 12.866 17.069 23.192 31.443 38.558 50.156 62.948
## [11] 75.995 91.972
## Data variable  tt : [1]  1  2  3  4  5  6  7  8  9 10 11 12
## trjfn:
## function (prm)
## {
##     if (is.null(names(prm)))
##         names(prm) <- names(pvec)
##     localdata <- list2env(as.list(prm), parent = data)
##     eval(residexpr, envir = localdata)
## }
## <bytecode: 0x55c43b156848>
## <environment: 0x55c43c2a3f28>
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Start:lamda: 1e-04  SS= 23521  at  b1 = 1  b2 = 1  b3 = 1  1 / 0
## lamda: 0.001  SS= 24356  at  b1 = 50.886  b2 = -240.72  b3 = -372.91  2 / 1
## lamda: 0.01  SS= 24356  at  b1 = 50.183  b2 = -190.69  b3 = -334.2  3 / 1
## lamda: 0.1  SS= 24356  at  b1 = 46.97  b2 = -25.309  b3 = -194.81  4 / 1
## lamda: 1  SS= 24356  at  b1 = 38.096  b2 = 29.924  b3 = -64.262  5 / 1
## lamda: 10  SS= 24353  at  b1 = 18.798  b2 = 3.551  b3 = -2.9011  6 / 1
## <<lamda: 4  SS= 21065  at  b1 = 4.1015  b2 = 0.86688  b3 = 1.3297  7 / 1
## <<lamda: 1.6  SS= 16852  at  b1 = 10.248  b2 = 1.2302  b3 = 1.0044  8 / 2
## lamda: 16  SS= 24078  at  b1 = 20.944  b2 = 2.9473  b3 = -0.40959  9 / 3
## <<lamda: 6.4  SS= 15934  at  b1 = 11.771  b2 = 1.3084  b3 = 0.98087  10 / 3
## <<lamda: 2.56  SS= 14050  at  b1 = 15.152  b2 = 1.6468  b3 = 0.80462  11 / 4
## <<lamda: 1.024  SS= 10985  at  b1 = 22.005  b2 = 2.6632  b3 = 0.45111  12 / 5
## <<lamda: 0.4096  SS= 6427.1  at  b1 = 35.128  b2 = 4.7135  b3 = 0.50974  13 / 6
## <<lamda: 0.16384  SS= 4725  at  b1 = 52.285  b2 = 9.2948  b3 = 0.31348  14 / 7
## <<lamda: 0.065536  SS= 1132.2  at  b1 = 76.446  b2 = 15.142  b3 = 0.41095  15 / 8
## <<lamda: 0.026214  SS= 518.76  at  b1 = 96.924  b2 = 24.422  b3 = 0.35816  16 / 9
## <<lamda: 0.010486  SS= 79.464  at  b1 = 122.18  b2 = 35.262  b3 = 0.36484  17 / 10
## <<lamda: 0.0041943  SS= 26.387  at  b1 = 141.55  b2 = 42.387  b3 = 0.35215  18 / 11
## <<lamda: 0.0016777  SS= 11.339  at  b1 = 160.02  b2 = 45.519  b3 = 0.33738  19 / 12
## <<lamda: 0.00067109  SS= 5.2767  at  b1 = 176.92  b2 = 47.037  b3 = 0.32443  20 / 13
```

```
## <<lamda: 0.00026844  SS= 2.9656  at  b1 = 189.12  b2 = 48.279  b3 = 0.31712  21 / 14
## <<lamda: 0.00010737  SS= 2.6003  at  b1 = 194.72  b2 = 48.92  b3 = 0.31429  22 / 15
## <<lamda: 4.295e-05  SS= 2.5873  at  b1 = 196.04  b2 = 49.075  b3 = 0.31364  23 / 16
## <<lamda: 1.718e-05  SS= 2.5873  at  b1 = 196.18  b2 = 49.091  b3 = 0.31357  24 / 17
## <<lamda: 6.8719e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  25 / 18
## <<lamda: 2.7488e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  26 / 19
## <<lamda: 1.0995e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  27 / 20
```

```
summary(anlxb1)
```

```
## $residuals
##  [1]  0.011900 -0.032755  0.092030  0.208782  0.392634 -0.057594 -1.105728
##  [8]  0.715786 -0.107648 -0.348396  0.652593 -0.287568
## attr(,"gradient")
##             b1       b2       b3
##  [1,] 0.027117 -0.10543   5.1756
##  [2,] 0.036737 -0.14142  13.8849
##  [3,] 0.049596 -0.18837  27.7424
##  [4,] 0.066645 -0.24858  48.8137
##  [5,] 0.089005 -0.32404  79.5373
##  [6,] 0.117921 -0.41568 122.4383
##  [7,] 0.154635 -0.52241 179.5225
##  [8,] 0.200186 -0.63986 251.2937
##  [9,] 0.255106 -0.75941 335.5263
## [10,] 0.319083 -0.86828 426.2517
## [11,] 0.390688 -0.95133 513.7254
## [12,] 0.467334 -0.99482 586.0466
##
## $sigma
## [1] 0.53617
##
## $df
## [1] 3 9
##
## $cov.unscaled
##           b1        b2          b3
## b1 444.72299 47.835371 -0.25280589
## b2  47.83537  9.916742 -0.01762908
## b3  -0.25281 -0.017629  0.00016386
##
## $param
##     Estimate Std. Error t value   Pr(>|t|)
## b1 196.18626 11.3069385  17.351 3.1667e-08
## b2  49.09164  1.6884366  29.075 3.2836e-10
## b3   0.31357  0.0068633  45.688 5.7676e-12
##
## $resname
## [1] "anlxb1"
##
## $ssquares
## [1] 2.5873
##
## $nobs
## [1] 12
##
```

```
## $ct
## [1] " " " " " " "
##
## $mt
## [1] " " " " " " "
##
## $Sd
## [1] 1.0108e+03 4.6047e-01 4.7144e-02
##
## $gr
##           [,1]
## b1 -8.8030e-12
## b2 -4.0610e-12
## b3  2.8321e-09
##
## $jeval
## [1] 20
##
## $feval
## [1] 27
##
## attr(,"pkgname")
## [1] "nlsr"
## attr(,"class")
## [1] "summary.nlsr"
```

print.nlsr Print summary output (but involving some serious computations!) of an object of class nlsr from `nlxb` or `nlfb` from package `nlsr`.

```
### From examples above
print(weedux)
```

```
## nlsr object: x
## residual sumsquares =  2.5873  on  12 observations
##     after  6    Jacobian and  7 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1              196.186         11.31       17.35  3.167e-08  -1.334e-10        1011
## b2              49.0916         1.688       29.08  3.284e-10  -3.589e-09      0.4605
## b3              0.31357      0.006863       45.69  5.768e-12    4.09e-07     0.04714
```

```
print(ans1)
```

```
## nlsr object: x
## residual sumsquares =  2.5873  on  12 observations
##     after  12    Jacobian and  16 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1              1.96186        0.1131       17.35  3.167e-08  -3.958e-08       130.1
## b2              4.90916        0.1688       29.08  3.284e-10   -6.93e-08       6.165
## b3               3.1357       0.06863       45.69  5.768e-12    8.84e-08       2.735
```

resgr, resss For a nonlinear model originally expressed as an expression of the form lhs ~ formula_for_rhs assume we have a resfn and jacfn that compute the residuals and the Jacobian at a set of parameters. This routine computes the gradient, that is, t(Jacobian) %*% residuals.

```
## Use shobbs example
RG <- resgr(st, shobbs.res, shobbs.jac)
```

```
RG
```

```
## [1] -10091.3   7835.3  -8234.2
```

```
SS <- resss(st, shobbs.res)
SS
```

```
## [1] 10685
```

####nlsSimplify and related functions####

Simplifications to render derivative expresssions more usable.

The related tools are: **newSimplification, sysSimplifications, isFALSE, isZERO, isONE, isMINUSONE, isCALL, findSubexprs, sysDerivs**.

nlsSimplify simplifies expressions according to rules specified by **newSimplification**.

findSubexprs finds common subexpressions in an expression vector so that duplicate computation can be avoided.

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

```
## a
```

```
nlsSimplify(quote(exp(1)), verbose = TRUE)
```

```
## Simplifying exp(1)
```

```
## Applying simplification:
```

```
## $expr
## exp(a)
##
## $argnames
## [1] "a"
##
## $test
## is.numeric(a)
##
## $simplification
## exp(a)
##
## $do_eval
## [1] TRUE
```

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

```
nlsSimplify(quote(sqrt(a + b)))  # standard rule
```

```
## sqrt(a + b)
```

```
## sysSimplifications
# creates a new environment whose parent is emptyenv()  Why??
str(sysSimplifications)
```

```
## <environment: 0x55c43bb97480>
```

```
myrules <- new.env(parent = sysSimplifications)
## newSimplification
```

```
newSimplification(sqrt(a), TRUE, a^0.5, simpEnv = myrules)
nlsSimplify(quote(sqrt(a + b)), simpEnv = myrules)
```

```
## (a + b)^0.5
```

```
## isFALSE
print(isFALSE(1==2))
```

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

```
print(isFALSE(2==2))
```

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

```
## isZERO
print(isZERO(0))
```

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

```
x <- -0
print(isZERO(x))
```

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

```
x <- 0
print(isZERO(x))
```

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

```
print(isZERO(~(-1)))
```

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

```
print(isZERO("-1"))
```

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

```
print(isZERO(expression(-1)))
```

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

```
## isONE
print(isONE(1))
```

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

```
x <- 1
print(isONE(x))
```

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

```
print(isONE(~(1)))
```

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

```
print(isONE("1"))
```

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

```
print(isONE(expression(1)))
```

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

```
## isMINUSONE
print(isMINUSONE(-1))
```

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

```
x <- -1
print(isMINUSONE(x))
```

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

```
print(isMINUSONE(~(-1)))
```

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

```
print(isMINUSONE("-1"))
```

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

```
print(isMINUSONE(expression(-1)))
```

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

```
## isCALL  ?? don't have good understanding of this
x <- -1
print(isCALL(x,"isMINUSONE"))
```

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

```
print(isCALL(x, quote(isMINUSONE)))
```

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

```
## findSubexprs
findSubexprs(expression(x^2, x-y, y^2-x^2))
```

```
## {
##     .expr1 <- x^2
##     expression(.expr1, x - y, y^2 - .expr1)
## }
```

```
## sysDerivs
# creates a new environment whose parent is emptyenv()  Why??
str(sysDerivs)
```

```
## <environment: 0x55c43da8b310>
```

summary.nlsr Provide a summary output (but involving some serious computations!) of an object of class nlsr from nlxb or nlfb from package nlsr. Examples have been given above under **nlxb** and **nlfb**.

wrapnlsr Given a nonlinear model expressed as an expression of the form

```
lhs ~ formula_for_rhs
```

and a start vector where parameters used in the model formula are named, attempts to find the minimum of the residual sum of squares using the Nash variant (Nash, 1979) of the Marquardt algorithm, where the linear sub-problem is solved by a qr method.

```
# Data for Hobbs problem
ydat  <-  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) # for testing
tdat  <-  seq_along(ydat) # for testing
```

```
start1  <-  c(b1=1, b2=1, b3=1)
eunsc  <-   y ~ b1/(1+b2*exp(-b3*tt))
weeddata1  <-  data.frame(y=ydat, tt=tdat)
## The following attempt with nls() fails!
anls1  <-  try(nls(eunsc, start=start1, trace=TRUE, data=weeddata1))
```

```
## 23521.    (2.08e+00): par = (1 1 1)
## Error in nls(eunsc, start = start1, trace = TRUE, data = weeddata1) :
##   singular gradient
```

```
## But we succeed by calling nlxb first.
anlxb1  <-  try(nlxb(eunsc, start=start1, trace=TRUE, data=weeddata1))
```

```
## formula: y ~ b1/(1 + b2 * exp(-b3 * tt))
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## $watch
## [1] FALSE
##
## $phi
## [1] 1
##
## $lamda
## [1] 1e-04
##
## $offset
## [1] 100
##
## $laminc
## [1] 10
##
## $lamdec
## [1] 4
##
## $femax
## [1] 10000
##
## $jemax
## [1] 5000
##
## $rofftest
## [1] TRUE
##
## $smallsstest
## [1] TRUE
##
## Finished masks check
## datvar:[1] "y"  "tt"
## Data variable  y : [1]  5.308  7.240  9.638 12.866 17.069 23.192 31.443 38.558 50.156 62.948
## [11] 75.995 91.972
## Data variable  tt : [1]  1  2  3  4  5  6  7  8  9 10 11 12
## trjfn:
## function (prm)
## {
```

```
##     if (is.null(names(prm)))
##         names(prm) <- names(pvec)
##     localdata <- list2env(as.list(prm), parent = data)
##     eval(residexpr, envir = localdata)
## }
## <bytecode: 0x55c43b156848>
## <environment: 0x55c43d3bee10>
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Start:lamda: 1e-04  SS= 23521  at  b1 = 1  b2 = 1  b3 = 1  1 / 0
## lamda: 0.001  SS= 24356  at  b1 = 50.886  b2 = -240.72  b3 = -372.91  2 / 1
## lamda: 0.01  SS= 24356  at  b1 = 50.183  b2 = -190.69  b3 = -334.2  3 / 1
## lamda: 0.1  SS= 24356  at  b1 = 46.97  b2 = -25.309  b3 = -194.81  4 / 1
## lamda: 1  SS= 24356  at  b1 = 38.096  b2 = 29.924  b3 = -64.262  5 / 1
## lamda: 10  SS= 24353  at  b1 = 18.798  b2 = 3.551  b3 = -2.9011  6 / 1
## <<lamda: 4  SS= 21065  at  b1 = 4.1015  b2 = 0.86688  b3 = 1.3297  7 / 1
## <<lamda: 1.6  SS= 16852  at  b1 = 10.248  b2 = 1.2302  b3 = 1.0044  8 / 2
## lamda: 16  SS= 24078  at  b1 = 20.944  b2 = 2.9473  b3 = -0.40959  9 / 3
## <<lamda: 6.4  SS= 15934  at  b1 = 11.771  b2 = 1.3084  b3 = 0.98087  10 / 3
## <<lamda: 2.56  SS= 14050  at  b1 = 15.152  b2 = 1.6468  b3 = 0.80462  11 / 4
## <<lamda: 1.024  SS= 10985  at  b1 = 22.005  b2 = 2.6632  b3 = 0.45111  12 / 5
## <<lamda: 0.4096  SS= 6427.1  at  b1 = 35.128  b2 = 4.7135  b3 = 0.50974  13 / 6
## <<lamda: 0.16384  SS= 4725  at  b1 = 52.285  b2 = 9.2948  b3 = 0.31348  14 / 7
## <<lamda: 0.065536  SS= 1132.2  at  b1 = 76.446  b2 = 15.142  b3 = 0.41095  15 / 8
## <<lamda: 0.026214  SS= 518.76  at  b1 = 96.924  b2 = 24.422  b3 = 0.35816  16 / 9
## <<lamda: 0.010486  SS= 79.464  at  b1 = 122.18  b2 = 35.262  b3 = 0.36484  17 / 10
## <<lamda: 0.0041943  SS= 26.387  at  b1 = 141.55  b2 = 42.387  b3 = 0.35215  18 / 11
## <<lamda: 0.0016777  SS= 11.339  at  b1 = 160.02  b2 = 45.519  b3 = 0.33738  19 / 12
## <<lamda: 0.00067109  SS= 5.2767  at  b1 = 176.92  b2 = 47.037  b3 = 0.32443  20 / 13
## <<lamda: 0.00026844  SS= 2.9656  at  b1 = 189.12  b2 = 48.279  b3 = 0.31712  21 / 14
## <<lamda: 0.00010737  SS= 2.6003  at  b1 = 194.72  b2 = 48.92  b3 = 0.31429  22 / 15
## <<lamda: 4.295e-05  SS= 2.5873  at  b1 = 196.04  b2 = 49.075  b3 = 0.31364  23 / 16
## <<lamda: 1.718e-05  SS= 2.5873  at  b1 = 196.18  b2 = 49.091  b3 = 0.31357  24 / 17
## <<lamda: 6.8719e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  25 / 18
## <<lamda: 2.7488e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  26 / 19
## <<lamda: 1.0995e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  27 / 20
```

```
st2 <- coef(anlxb1)
anls2  <-  try(nls(eunsc, start=st2, trace=TRUE, data=weeddata1))
```

```
## 2.5873    (6.57e-08): par = (196.19 49.092 0.31357)
```

```
## Or we can simply call wrapnlsr
anls2a  <-  try(wrapnlsr(eunsc, start=start1, trace=TRUE, data=weeddata1))
```

```
## wrapnls call with lower=[1] -Inf -Inf -Inf
## and upper=[1] Inf Inf Inf
## formula: y ~ b1/(1 + b2 * exp(-b3 * tt))
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## $watch
## [1] FALSE
##
## $phi
```

```
## [1] 1
##
## $lamda
## [1] 1e-04
##
## $offset
## [1] 100
##
## $laminc
## [1] 10
##
## $lamdec
## [1] 4
##
## $femax
## [1] 10000
##
## $jemax
## [1] 5000
##
## $rofftest
## [1] TRUE
##
## $smallsstest
## [1] TRUE
##
## Finished masks check
## datvar:[1] "y"  "tt"
## Data variable  y : [1]  5.308  7.240  9.638 12.866 17.069 23.192 31.443 38.558 50.156 62.948
## [11] 75.995 91.972
## Data variable  tt : [1]  1  2  3  4  5  6  7  8  9 10 11 12
## trjfn:
## function (prm)
## {
##     if (is.null(names(prm)))
##         names(prm) <- names(pvec)
##     localdata <- list2env(as.list(prm), parent = data)
##     eval(residexpr, envir = localdata)
## }
## <bytecode: 0x55c43b156848>
## <environment: 0x55c43c4af168>
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Start:lamda: 1e-04  SS= 23521  at  b1 = 1  b2 = 1  b3 = 1  1 / 0
## lamda: 0.001  SS= 24356  at  b1 = 50.886  b2 = -240.72  b3 = -372.91  2 / 1
## lamda: 0.01  SS= 24356  at  b1 = 50.183  b2 = -190.69  b3 = -334.2  3 / 1
## lamda: 0.1  SS= 24356  at  b1 = 46.97  b2 = -25.309  b3 = -194.81  4 / 1
## lamda: 1  SS= 24356  at  b1 = 38.096  b2 = 29.924  b3 = -64.262  5 / 1
## lamda: 10  SS= 24353  at  b1 = 18.798  b2 = 3.551  b3 = -2.9011  6 / 1
## <<lamda: 4  SS= 21065  at  b1 = 4.1015  b2 = 0.86688  b3 = 1.3297  7 / 1
## <<lamda: 1.6  SS= 16852  at  b1 = 10.248  b2 = 1.2302  b3 = 1.0044  8 / 2
## lamda: 16  SS= 24078  at  b1 = 20.944  b2 = 2.9473  b3 = -0.40959  9 / 3
## <<lamda: 6.4  SS= 15934  at  b1 = 11.771  b2 = 1.3084  b3 = 0.98087  10 / 3
```

```
## <<lamda: 2.56  SS= 14050  at  b1 = 15.152  b2 = 1.6468  b3 = 0.80462  11 / 4
## <<lamda: 1.024  SS= 10985  at  b1 = 22.005  b2 = 2.6632  b3 = 0.45111  12 / 5
## <<lamda: 0.4096  SS= 6427.1  at  b1 = 35.128  b2 = 4.7135  b3 = 0.50974  13 / 6
## <<lamda: 0.16384  SS= 4725  at  b1 = 52.285  b2 = 9.2948  b3 = 0.31348  14 / 7
## <<lamda: 0.065536  SS= 1132.2  at  b1 = 76.446  b2 = 15.142  b3 = 0.41095  15 / 8
## <<lamda: 0.026214  SS= 518.76  at  b1 = 96.924  b2 = 24.422  b3 = 0.35816  16 / 9
## <<lamda: 0.010486  SS= 79.464  at  b1 = 122.18  b2 = 35.262  b3 = 0.36484  17 / 10
## <<lamda: 0.0041943  SS= 26.387  at  b1 = 141.55  b2 = 42.387  b3 = 0.35215  18 / 11
## <<lamda: 0.0016777  SS= 11.339  at  b1 = 160.02  b2 = 45.519  b3 = 0.33738  19 / 12
## <<lamda: 0.00067109  SS= 5.2767  at  b1 = 176.92  b2 = 47.037  b3 = 0.32443  20 / 13
## <<lamda: 0.00026844  SS= 2.9656  at  b1 = 189.12  b2 = 48.279  b3 = 0.31712  21 / 14
## <<lamda: 0.00010737  SS= 2.6003  at  b1 = 194.72  b2 = 48.92  b3 = 0.31429  22 / 15
## <<lamda: 4.295e-05  SS= 2.5873  at  b1 = 196.04  b2 = 49.075  b3 = 0.31364  23 / 16
## <<lamda: 1.718e-05  SS= 2.5873  at  b1 = 196.18  b2 = 49.091  b3 = 0.31357  24 / 17
## <<lamda: 6.8719e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  25 / 18
## <<lamda: 2.7488e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  26 / 19
## <<lamda: 1.0995e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  27 / 20
## nlsr object: x
## residual sumsquares =  2.5873  on  12 observations
##     after  20    Jacobian and  27 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               196.186         11.31      17.35  3.167e-08  -8.803e-12        1011
## b2               49.0916         1.688      29.08  3.284e-10  -4.061e-12      0.4605
## b3               0.31357      0.006863      45.69  5.768e-12   2.832e-09     0.04714
## newstart:       b1        b2        b3
## 196.18626  49.09164   0.31357
## nls call with no bounds
## 2.5873    (6.57e-08): par = (196.19 49.092 0.31357)
```

Particular features

weighted nonlinear regression

Weighted regression alters the sum of squares of the parameters `prm` from

```
sum_{i} (residual_i(prm)^2)
```

to

```
sum_{i} (wts_i * residual_i(prm)^2)
```

where `wts` is the vector of weights. This is equivalent to multiplying each residual or row of the Jacobian by the square root of the respective weight.

While `nls()` has provision for (fixed) weights, the example given in the documentation of `nls()` for weighted regression actually uses a functional form. Moreover, the weights are the square roots of the **predicted** or model values, so are not fixed. Here we replace these weights with the fixed values of the quantity we are trying to predict, namely the `rate` variable in a kinetic model. In our example below we first use the documentation example with predicted values; note that the name of the result has been changed from that in the `nls()` documentation. This result gives different estimates of the parameters from the fixed weights used afterwards.

Note that neither `nlsr::nlxb` nor `nlsr::nlfb` can use the weighted.MM function directly. This is one of the more awkward aspects of nonlinear least squares and nonlinear optimization.

```
Treated <- Puromycin[Puromycin$state == "treated", ]
weighted.MM <- function(resp, conc, Vm, K)
{
```

```
  ## Purpose: exactly as white book p. 451 -- RHS for nls()
  ##  Weighted version of Michaelis-Menten model
  ## ------------------------------------------------------------
  ## Arguments: 'y', 'x' and the two parameters (see book)
  ## ------------------------------------------------------------
  ## Author: Martin Maechler, Date: 23 Mar 2001

  pred <- (Vm * conc)/(K + conc)
  (resp - pred) / sqrt(pred)
}
## Here is the estimation using predicted value weights
Pur.wtnlspred <- nls( ~ weighted.MM(rate, conc, Vm, K), data = Treated,
               start = list(Vm = 200, K = 0.1))
summary(Pur.wtnlspred)
```

```
##
## Formula: 0 ~ weighted.MM(rate, conc, Vm, K)
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## Vm 2.07e+02   9.22e+00   22.42  7.0e-10 ***
## K  5.46e-02   7.98e-03    6.84  4.5e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.21 on 10 degrees of freedom
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 3.86e-06
```

```
## nlxb cannot use this form
Pur.wtnlxbpred <- try(nlxb( ~ weighted.MM(rate, conc, Vm, K), data = Treated,
               start = list(Vm = 200, K = 0.1)))
```

```
## Error in deriv.default(residexpr, names(pvec)) :
##   Function 'weighted.MM' is not in the derivatives table
```

```
## and the structure is wrong for nlfb. See wres.MM below.
Pur.wtnlfbpred <- try(nlfb(resfn=weighted.MM(rate, conc, Vm, K), data = Treated,
               start = list(Vm = 200, K = 0.1)))
```

```
## Error in weighted.MM(rate, conc, Vm, K) : object 'Vm' not found
```

```
## Now use fixed weights and the weights argument in nls()
Pur.wnls <- nls( rate ~ (Vm * conc)/(K + conc), data = Treated,
                 start = list(Vm = 200, K = 0.1), weights=1/rate)
summary(Pur.wnls)
```

```
##
## Formula: rate ~ (Vm * conc)/(K + conc)
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## Vm 2.10e+02   9.01e+00   23.27  4.9e-10 ***
## K  6.07e-02   8.39e-03    7.23  2.8e-05 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.11 on 10 degrees of freedom
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 9.48e-07
```

```
## nlsr::nlxb should give essentially the same result
library(nlsr)
## Note that we put in the dataframe name to explicitly specify weights
Pur.wnlxb <- nlxb( rate ~ (Vm * conc)/(K + conc), data = Treated,
               start = list(Vm = 200, K = 0.1), weights=1/Treated$rate)
summary(Pur.wnlxb)
```

```
## $residuals
##  [1] -2.755920  0.725597  0.734149 -0.267735  1.091190 -0.330634  0.420283
##  [8]  0.997627 -0.136479 -0.838386 -0.580807 -0.095909
## attr(,"gradient")
##            Vm       K
##  [1,] 0.24797 -644.41
##  [2,] 0.24797 -644.41
##  [3,] 0.49729 -863.88
##  [4,] 0.49729 -863.88
##  [5,] 0.64458 -791.67
##  [6,] 0.64458 -791.67
##  [7,] 0.78388 -585.42
##  [8,] 0.78388 -585.42
##  [9,] 0.90227 -304.70
## [10,] 0.90227 -304.70
## [11,] 0.94774 -171.15
## [12,] 0.94774 -171.15
##
## $sigma
## [1] 1.1078
##
## $df
## [1]  2 10
##
## $cov.unscaled
##           Vm          K
## Vm 66.089006 4.7937e-02
## K   0.047937 5.7385e-05
##
## $param
##      Estimate Std. Error t value   Pr(>|t|)
## Vm 209.596815  9.0058767 23.2733 4.8540e-10
## K    0.060654  0.0083919  7.2276 2.8322e-05
##
## $resname
## [1] "Pur.wnlxb"
##
## $ssquares
## [1] 12.272
##
## $nobs
```

```
## [1] 12
##
## $ct
## [1] " " " " "
##
## $mt
## [1] " " " " "
##
## $Sd
## [1] 210.28204   0.12301
##
## $gr
##           [,1]
## Vm -1.3390e-13
## K  -8.8003e-12
##
## $jeval
## [1] 9
##
## $feval
## [1] 10
##
## attr(,"pkgname")
## [1] "nlsr"
## attr(,"class")
## [1] "summary.nlsr"
```

```
## check difference
coef(Pur.wnls) - coef(Pur.wnlxb)
```

```
##          Vm           K
## -2.2903e-05 -2.7428e-08
## attr(,"pkgname")
## [1] "nlsr"
```

```
## Residuals Pur.wnls -- These are weighted
## resid(Pur.wnlxb) # ?? does not work -- why?
print(res(Pur.wnlxb))
```

```
##  [1] -2.755920  0.725597  0.734149 -0.267735  1.091190 -0.330634  0.420283
##  [8]  0.997627 -0.136479 -0.838386 -0.580807 -0.095909
## attr(,"gradient")
##            Vm       K
##  [1,] 0.24797 -644.41
##  [2,] 0.24797 -644.41
##  [3,] 0.49729 -863.88
##  [4,] 0.49729 -863.88
##  [5,] 0.64458 -791.67
##  [6,] 0.64458 -791.67
##  [7,] 0.78388 -585.42
##  [8,] 0.78388 -585.42
##  [9,] 0.90227 -304.70
## [10,] 0.90227 -304.70
## [11,] 0.94774 -171.15
## [12,] 0.94774 -171.15
```

```
## Those from nls() are NOT weighted
resid(Pur.wnls)
```

```
##  [1]  24.0255  -4.9745  -7.2305   2.7695 -12.1019   3.8981  -5.2996 -12.2996
##  [9]   1.8862  11.8862   8.3564   1.3564
## attr(,"label")
## [1] "Residuals"
```

```
## Try using a function, that is nlsr::nlfb
stw <- c(Vm = 200, K = 0.1)
wres.MM <- function(prm, rate, conc) {
      Vm <- prm[1]
      K <- prm[2]
      pred <- (Vm * conc)/(K + conc)
    (rate - pred) / sqrt(rate) # NOTE: NOT pred here
}

# test function first to see it works
print(wres.MM(stw, rate=Treated$rate, conc=Treated$conc))
```

```
##  [1] 4.8942 1.9935 2.2338 3.0936 1.6445 2.9040 1.7051 1.1761 1.5414 2.2079
## [11] 1.6449 1.1785
```

```
Pur.wnlfb <- nlfb(start=stw,  resfn=wres.MM, rate = Treated$rate, conc=Treated$conc,
                  trace=FALSE) # Note: weights are inside function already here
summary(Pur.wnlfb)
```

```
## $residuals
##  [1]  2.755920 -0.725597 -0.734149  0.267735 -1.091190  0.330634 -0.420283
##  [8] -0.997627  0.136479  0.838386  0.580807  0.095909
##
## $sigma
## [1] 1.1078
##
## $df
## [1]  2 10
##
## $cov.unscaled
##           Vm          K
## Vm 66.089009 4.7937e-02
## K   0.047937 5.7385e-05
##
## $param
##      Estimate Std. Error t value   Pr(>|t|)
## Vm 209.596815  9.0058769 23.2733 4.8540e-10
## K    0.060654  0.0083919  7.2276 2.8322e-05
##
## $resname
## [1] "Pur.wnlfb"
##
## $ssquares
## [1] 12.272
##
## $nobs
## [1] 12
```

```
##
## $ct
## [1] " " " " "
##
## $mt
## [1] " " " " "
##
## $Sd
## [1] 210.28202   0.12301
##
## $gr
##             [,1]
## [1,] -1.3798e-13
## [2,] -2.8741e-12
##
## $jeval
## [1] 9
##
## $feval
## [1] 10
##
## attr(,"pkgname")
## [1] "nlsr"
## attr(,"class")
## [1] "summary.nlsr"
```

```
print(Pur.wnlfb)
```

```
## nlsr object: x
## residual sumsquares =  12.272  on  12 observations
##     after  9    Jacobian and  10 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## Vm               209.597         9.006      23.27  4.854e-10   -1.38e-13       210.3
## K              0.0606538      0.008392      7.228  2.832e-05  -2.874e-12       0.123
```

```
## check estimates with nls() result
coef(Pur.wnls) - coef(Pur.wnlfb)
```

```
##          Vm           K
## -2.3153e-05 -2.7733e-08
## attr(,"pkgname")
## [1] "nlsr"
```

```
## and with nlxb result
coef(Pur.wnlxb) - coef(Pur.wnlfb)
```

```
##          Vm           K
## -2.5003e-07 -3.0479e-10
## attr(,"pkgname")
## [1] "nlsr"
```

```
wres0.MM <- function(prm, rate, conc) {
      Vm <- prm[1]
      K <- prm[2]
      pred <- (Vm * conc)/(K + conc)
    (rate - pred) # NOTE: NO weights
}
```

```
Pur.wnlfb0 <- nlfb(start=stw,  resfn=wres0.MM, rate = Treated$rate, conc=Treated$conc,
                  weights=1/Treated$rate, trace=FALSE) # Note: explicit weights
summary(Pur.wnlfb0)
```

```
## $residuals
##  [1] 4.8942 1.9935 2.2338 3.0936 1.6445 2.9040 1.7051 1.1761 1.5414 2.2079
## [11] 1.6449 1.1785
##
## $sigma
## [1] 2.6317
##
## $df
## [1]  2 10
##
## $cov.unscaled
##           Vm           K
## Vm 70.607853 -3.5304e-02
## K  -0.035304  1.7652e-05
##
## $param
##    Estimate Std. Error t value   Pr(>|t|)
## Vm    200.0  22.114175   9.044 3.9605e-06
## K       0.1   0.011057   9.044 3.9606e-06
##
## $resname
## [1] "Pur.wnlfb0"
##
## $ssquares
## [1] 69.261
##
## $nobs
## [1] 12
##
## $ct
## [1] " " " " "
##
## $mt
## [1] " " " " "
##
## $Sd
## [1] 7.4995e+08 1.1901e-01
##
## $gr
##            [,1]
## [1,]    3119894
## [2,] 6239784883
##
## $jeval
## [1] 1
##
## $feval
## [1] 14
##
## attr(,"pkgname")
```

```
## [1] "nlsr"
## attr(,"class")
## [1] "summary.nlsr"
```

```
print(Pur.wnlfb0)
```

```
## nlsr object: x
## residual sumsquares =  69.261  on  12 observations
##     after  1    Jacobian and  14 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## Vm                  200         22.11       9.044  3.961e-06     3119894   749947494
## K                   0.1       0.01106       9.044  3.961e-06    6.24e+09       0.119
```

```
## check estimates with nls() result
coef(Pur.wnls) - coef(Pur.wnlfb0)
```

```
##        Vm         K
##  9.596792 -0.039346
## attr(,"pkgname")
## [1] "nlsr"
```

```
## and with nlxb result
coef(Pur.wnlxb) - coef(Pur.wnlfb0)
```

```
##        Vm         K
##  9.596815 -0.039346
## attr(,"pkgname")
## [1] "nlsr"
```

```
wss.MM <- function(prm, rate, conc) {
  ss <- as.numeric(crossprod(wres.MM(prm, rate, conc)))
}
library(optimx)
osol <- optimr(stw, fn=wss.MM, gr="grnd", method="Nelder-Mead", control=list(trace=0),
            rate=Treated$rate, conc=Treated$conc)
print(osol)
```

```
## $par
##         Vm          K
## 209.596996   0.060653
##
## $value
## [1] 12.272
##
## $counts
## function gradient
##       69       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
## difference from nlxb
osol$par - coef(Pur.wnlxb)
```

```
##          Vm           K
```

```
##  1.8037e-04 -1.2162e-06
## attr(,"pkgname")
## [1] "nlsr"
```

How QR decomposition is used to effect the Levenberg-Marquardt stabilization

If the nonlinear least squares problem is defined by minimizing the sum of squares of a vector of residuals that are functions of a vector of parameters `x`

```
r =  f(x)
```

Then the minimization of `S(x) = r^T r` is our goal.

`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 now (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. (Nash (1979)), 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 `\lambda` is reduced. If not, we increase the size of `\lambda` and compute a new `\delta`. 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).

We could implement the methods using the equations above. However, the accumulation of inner products in `J^T J` occasions some numerical error, and it is generally both safer and more efficient to use matrix decompositions. In particular, if we form the QR decomposition of `J`

```
`Q R = J`
```

where Q is an orthonormal matrix and R is Right or Upper triangular, we can easily solve

```
R \delta = Q^T r
```

But how do we get the Marquardt stabilization?

If we augment `J` with a square matrix `sqrt(\lambda D)` whose diagonal elements are the square roots of `\lambda` times the diagonal elements of `D`, and augment the vector `r` with `n` zeros where `n` is the column dimension of `J` and `D`, we achieve our goal.

Typically we can use `D = 1_n` (the identidy of order n), but Marquardt (1963) showed that using the diagonal elements of `J^T J` for D. Nash (1977) pointed out that on computers with limited arithmetic (which now are rare since the IEEE 754 standard appeared in 1985), underflow might cause a problem of very small

elements in D and proposed adding \phi 1_n to the diagonals of J^T J before multiplying by \lambda in the Marquardt stabilization. This avoids some awkward cases with very little extra overhead. It is accomplished with the QR approach by appending sqrt(\phi * \lambda) times a unit matrix 1_n to the matrix already augmented matrix. We must also append n zeros to the augmented r.

Relative offset convergence criterion

In October 2020, I suggested a patch to the stats::nls() function to avoid a zero-divide in the relative offset convergence criterion. The explanation of this in the proposed manual file 'nls.Rd' is

"**Warning**

The default settings of nls generally fail on small-residual problems.

The nls function uses a relative-offset convergence criterion that compares the numerical imprecision at the current parameter estimates to the residual sum-of-squares. To avoid a zero-divide in computing the test value, a positive constant convTestAdd should be added to the sum-of-squares. This quantity is set via nls.control(), as in the example below."

Missing capabilities

Multi-line function expressions Multi-line functions present a challenge. This is in part because the chain rule may have to be applied backwards (last line first), but also because there may be structures that are not always differentiable, such as *if* statements.

Note that the functional approach to nonlinear least squares – accessible via function nlfb – does let us prepare residuals and Jacobians of complexity limited only by the capabilities of **R**.

Automatic differentiation This is the natural extension of Multi-line expressions. The authors would welcome collaboration with someone who has expertise in this area.

Vector parameters We would like to be able to find the Jacobian or gradient of functions that have as their parameters a vector, e.g., *prm*. At time of writing (January 2015) we cannot specify such a vector within **nlsr**. ?? is this true?

Examples of capabilities and weaknesses The following (rather long) script shows things that work or fail. In particular, it shows that the calls needed to get derivatives need to be set up precisely. This is not unique to R – the same sort of attention to (syntax specific) detail is present in other systems like Macsyma/Maxima.

First we set up the Hobbs weed problem. Initially it seemed a good idea to put the data WITHIN the residual function so that it would not have to be passed to the function. In retrospect, this is a bad idea because y and t are needed. To make the use of the data explicit, it is saved in the variables tdat for time and ydat for the weed data.

```
# tryhobbsderiv.R
ydat<-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)
tdat<-1:12
# try setting t and y here
t <- tdat
y <- ydat
# now define a function

hobbs.res<-function(x, t, y){ # Hobbs weeds problem -- residual
  # This variant uses looping
#  if(length(x) != 3) stop("hobbs.res -- parameter vector n!=3")
```

```
#  if(abs(12*x[3])>50) {
#    res<-rep(Inf,12)
#  } else {
    res<-x[1]/(1+x[2]*exp(-x[3]*t)) - y
#  }
}
# test it
start1 <- c(200, 50, .3)
## residuals at start1
r1 <- hobbs.res(start1, t=tdat, y=ydat)
print(r1)
```

```
##  [1] -0.050502 -0.207795 -0.260868 -0.412475 -0.616907 -1.605254 -3.364239
##  [8] -2.430165 -4.287340 -5.630791 -5.675428 -7.447795
```

Now we try some of the derivative capabilities of `nlsr`.

```
## NOTE: some functions may be seemingly correct for R, but we do not
## get the result desired, despite no obvious error. Always test.
require(nlsr)
# Try directly to differentiate the residual vector. r1 is numeric, so this should
# return a vector of zeros in a mathematical sense. In fact it gives an error,
# since R does not want to differentiate a numeric vector.
Jr1a <- fnDeriv(r1, "x")
```

```
## Error in codeDeriv(expr, namevec, do_substitute = FALSE, verbose = verbose, : Only single expressions
```

```
# Set up a function containing expression with subscripted parameters x[]
hobbs1 <- function(x, t, y){ res<-x[1]/(1+x[2]*exp(-x[3]*t)) - y }
# test the residuals
print(hobbs1(start1, t=tdat, y=ydat))
```

```
##  [1] -0.050502 -0.207795 -0.260868 -0.412475 -0.616907 -1.605254 -3.364239
##  [8] -2.430165 -4.287340 -5.630791 -5.675428 -7.447795
```

```
# Alternatively, let us set up t and y
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)
t<-1:12
# try different calls. Note that we need to include t and y data somehow
print(hobbs1(start1))
```

```
## Error in hobbs1(start1): argument "t" is missing, with no default
```

```
print(hobbs1(start1, t, y))
```

```
##  [1] -0.050502 -0.207795 -0.260868 -0.412475 -0.616907 -1.605254 -3.364239
##  [8] -2.430165 -4.287340 -5.630791 -5.675428 -7.447795
```

```
print(hobbs1(start1, tdat, ydat))
```

```
##  [1] -0.050502 -0.207795 -0.260868 -0.412475 -0.616907 -1.605254 -3.364239
##  [8] -2.430165 -4.287340 -5.630791 -5.675428 -7.447795
```

```
# We remove t and y, to ensure we don't get results from their presence
rm(t)
rm(y)
# Set up a function containing an expression with named (with numbers) parameters
# Note that we need to link these to the values in x
```

```
hobbs1m <- function(x, t, y){
  x001 <- x[1]
  x002 <- x[2]
  x003 <- x[3]
  res<-x001/(1+x002*exp(-x003*t)) - y
}
print(hobbs1m(x=start1, t=tdat, y=ydat))
```

```
##  [1] -0.050502 -0.207795 -0.260868 -0.412475 -0.616907 -1.605254 -3.364239
##  [8] -2.430165 -4.287340 -5.630791 -5.675428 -7.447795
```

```
# Function with explicit use of the expression()
hobbs1me <- function(x, t, y){
  x001 <- x[1]
  x002 <- x[2]
  x003 <- x[3]
  expression(x001/(1+x002*exp(-x003*t)) - y)
}
print(hobbs1me(start1, t=tdat, y=ydat))
```

```
## expression(x001/(1 + x002 * exp(-x003 * t)) - y)
```

```
# note failure (because the expression is not evaluated?)
#
# Now try to take derivatives
Jr11m <- fnDeriv(hobbs1m, c("x001", "x002", "x003"))
```

```
## Error in as.vector(x, "expression"): cannot coerce type 'closure' to vector of type 'expression'
```

```
# fails because expression is INSIDE a function (i.e., closure)
#
# try directly differentiating the expression
Jr11ex <-fnDeriv(expression(x001/(1+x002*exp(-x003*t)) - y)
                 , c("x001", "x002", "x003"))
# this seems to "work". Let us display the result
Jr11ex
```

```
## function (x001, x002, x003, t, y)
## {
##     .expr1 <- -x003
##     .expr2 <- .expr1 * t
##     .expr3 <- exp(.expr2)
##     .expr4 <- x002 * .expr3
##     .expr5 <- 1 + .expr4
##     .expr6 <- .expr5^2
##     .value <- x001/(.expr5) - y
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("x001",
##     "x002", "x003")))
##     .grad[, "x001"] <- 1/.expr5
##     .grad[, "x002"] <- -(x001 * .expr3/.expr6)
##     .grad[, "x003"] <- -(x001 * (x002 * (.expr3 * -t))/.expr6)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
# Set the values of the parameters by name
x001 <- start1[1]
x002 <- start1[2]
x003 <- start1[3]
# and try to evaluate
print(eval(Jr11ex))
```

```
## function (x001, x002, x003, t, y)
## {
##     .expr1 <- -x003
##     .expr2 <- .expr1 * t
##     .expr3 <- exp(.expr2)
##     .expr4 <- x002 * .expr3
##     .expr5 <- 1 + .expr4
##     .expr6 <- .expr5^2
##     .value <- x001/(.expr5) - y
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("x001",
##     "x002", "x003")))
##     .grad[, "x001"] <- 1/.expr5
##     .grad[, "x002"] <- -(x001 * .expr3/.expr6)
##     .grad[, "x003"] <- -(x001 * (x002 * (.expr3 * -t))/.expr6)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
# we need t and y, so set them
t<-tdat
y<-ydat
print(eval(Jr11ex))
```

```
## function (x001, x002, x003, t, y)
## {
##     .expr1 <- -x003
##     .expr2 <- .expr1 * t
##     .expr3 <- exp(.expr2)
##     .expr4 <- x002 * .expr3
##     .expr5 <- 1 + .expr4
##     .expr6 <- .expr5^2
##     .value <- x001/(.expr5) - y
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("x001",
##     "x002", "x003")))
##     .grad[, "x001"] <- 1/.expr5
##     .grad[, "x002"] <- -(x001 * .expr3/.expr6)
##     .grad[, "x003"] <- -(x001 * (x002 * (.expr3 * -t))/.expr6)
##     attr(.value, "gradient") <- .grad
##     .value
## }
```

```
# But there is still a problem. WHY???
#
# Let us try it piece by piece (column by column)
resx <- expression(x001/(1+x002*exp(-x003*t)) - y)
res1 <- Deriv(resx, "x001", do_substitute=FALSE)
```

```
## Error in Deriv(resx, "x001", do_substitute = FALSE): unused argument (do_substitute = FALSE)
```

```
res1
```

```
## Error in eval(expr, envir, enclos): object 'res1' not found
```

```
col1 <- eval(res1)
```

```
## Error in eval(res1): object 'res1' not found
```

```
res2 <- Deriv(resx, "x002", do_substitute=FALSE)
```

```
## Error in Deriv(resx, "x002", do_substitute = FALSE): unused argument (do_substitute = FALSE)
```

```
res2
```

```
## Error in eval(expr, envir, enclos): object 'res2' not found
```

```
col2 <- eval(res2)
```

```
## Error in eval(res2): object 'res2' not found
```

```
res3 <- Deriv(resx, "x003", do_substitute=FALSE)
```

```
## Error in Deriv(resx, "x003", do_substitute = FALSE): unused argument (do_substitute = FALSE)
```

```
res3
```

```
## Error in eval(expr, envir, enclos): object 'res3' not found
```

```
col3 <- eval(res3)
```

```
## Error in eval(res3): object 'res3' not found
```

```
hobJac <- cbind(col1, col2, col3)
```

```
## Error in cbind(col1, col2, col3): object 'col1' not found
```

```
print(hobJac)
```

```
## Error in print(hobJac): object 'hobJac' not found
```

Clearly there is still some work to do to make the mechanism for getting derivatives more easily, and with less chance of error. Work still to do???.

Jacobians

Jacobians, the matrices of partial derivatives of residuals with respect to the parameters, have a vector input (the parameters). `nlsr` attempts to generate the code for this computation. **This is the first key improvement of `nlsr` over `nls()`.** It will likely be a continuing effort to enlarge the set of functions and expressions that can be handled and to deal with them more efficiently. Also it would be nice to automate the generation of numerical approximations for those components of the Jacobian for which analytic derivatives are not available.

Note that the Jacobian inner product (J^T J) differs from the Hessian of the sum of squares by a matrix whose elements that are an inner product of the residuals with their second derivatives with respect to the parameters. This is a summation of m matrices each involving a residual times its (n by n) partial derivatives with respect to the parameters. Generally we treat this matrix as "ignorable" on the basis that the residuals should be "small", but clearly this is not always the case.

Implementation of nonlinear least squares methods

Gauss-Newton variants

Nonlinear least squares methods are mostly founded on some or other variant of the Gauss-Newton algorithm. The function we wish to minimize is the sum of squares of the (nonlinear) residuals r(x) where there are m

observations (elements of r) and n parameters x. Hence the function is

```
f(x) = sum(r(k)^2)
```

Newton's method starts with an original set of parameters x[0]. At a given iteraion, which could be the first, we want to solve

```
x[k+1]  =  x[k]  -   H^(-1) g
```

where H is the Hessian and g is the gradient at x[k]. We can rewrite this as a solution, at each iteration, of

```
H delta = -g
```

with

```
x[k+1]  =  x[k] + delta
```

For the particular sum of squares, the gradient is

```
g(x) = 2 * r(k)
```

and

```
H(x) = 2 ( J' J  +  sum(r * Z))
```

where J is the Jacobian (first derivatives of r w.r.t. x) and Z is the tensor of second derivatives of r w.r.t. x). Note that J' is the transpose of J.

The primary simplification of the Gauss-Newton method is to assume that the second term above is negligible. As there is a common factor of 2 on each side of the Newton iteration after the simplification of the Hessian, the Gauss-Newton iteration equation is

```
J' J  delta = - J' r
```

This iteration frequently fails. The approximation of the Hessian by the Jacobian inner-product is one reason, but there is also the possibility that the sum of squares function is not "quadratic" enough that the unit step reduces it. Hartley (1961) introduced a line search along delta, while Marquardt (1963) suggested replacing J' J with (J' J + lambda * D) where D is a diagonal matrix intended to partially approximate the omitted portion of the Hessian.

Marquardt suggested D = I (a unit matrix) or D = (diagonal part of J' J). The former approach, when lambda is large enough that the iteration is essentially

```
delta = - g / lambda
```

we get a version of the steepest descents algorithm. Using the diagonal of J' J, we have a scaled version of this (see https://en.wikipedia.org/wiki/Levenberg%E2%80%93Marquardt_algorithm)

Nash (1977) found that on low precision machines, it was common for diagonal elements of J' J to underflow. A very small modification to solve

```
(J' J + lambda * (D + phi * I)) * delta = - g
```

where phi is a small number (phi = 1 seems to work quite well) ?? check??.

In jncnm79, the iteration equation was solved as stated. However, this involves forming the sum of squares and cross products of J, a process that loses some numerical precision. A better way to solve the linear equations is to apply the QR decomposition to the matrix J itself. However, we still need to incorporate the lambda * I or lambda * D adjustments. This is done by adding rows to J that are the square roots of the "pieces". We add 1 row for each diagonal element of I and each diagonal element of D.

In each iteration, we reduce the lambda parameter before solution. If the resulting sum of squares is not reduced, lambda is increased, otherwise we move to the next iteration. Various authors (including the present one) have suggested different strategies for this. My current opinion is that a "quick" increase, say a factor of 10, and a "slow" decrease, say a factor of 0.2, work quite well. However, it is important to check that lambda has not got too small or underflowed before applying the increase factor. On the other hand, it is useful to be able to set lambda = 0 in the code so that a pure Gauss-Newton method can be evaluated with the program(s). The current code `nlfb()` uses the line

```
if (lamda<1000*.Machine$double.eps) lamda<-1000*.Machine$double.eps
```

to ensure we get an increase. To force a Gauss-Newton algorithm, the controls `laminc` and `lamdec` are set to 0.

The Levenberg-Marquardt adjustment to the Gauss-Newton approach is the second major improvement of `nlsr` (and also its predecessor `nlmrt` and the package `minpack-lm`) over `nls()`.

Termination and convergence tests

The success of many optimization codes often depends on knowing when no more progress can be made. For the present codes, one simple procedure increases the damping parameter lamda whenever the sum of squares cannot be decreased, with termination when the parameters are not altered by the addition of delta. While this "works", it does incur unnecessary computational effort.

A better approach is that of Bates and Watts (1981). Their relative offset criterion considers the potential progress that could be made in reducing the current residuals to the initial sum of squares. This is a sensible and very effective strategy for most situations, but is dangerous where the problem has very small or zero residuals, as in the case of an interpolation problem or nonlinear equations.

To illustrate this for zero residual problems, let us set up a simple test.

```
## test small resid case with roffset
tt <- 1:25
ymod <- 10 * exp(-0.01*tt) + 5
n <- length(tt)
evec0 <- rep(0, n)
evec1 <- 1e-4*runif(n, -.5, .5)
evec2 <-  1e-1*runif(n, -.5, .5)
y0 <- ymod + evec0
y1 <- ymod + evec1
y2 <- ymod + evec2
mydata <- data.frame(tt, y0, y1, y2)
st <- c(aa=1, bb=1, cc=1)
```

We provide three sizes of residuals here, but will only illustrate the consequences of zero residuals (the problem with y0). Let us run the nonlinear least squares problem to get the interpolating function, first with `nls()`, then with `nlxb()`. In the second case we have turned off the trace, as the approach "works" because we have taken care in the code to provide for problems with small residuals.

```
nlsfit0 <-  try(nls(y0 ~ aa * exp(-bb*tt) + cc, start=st, data=mydata, trace=TRUE))
```

```
## 4092.5    (2.74e+01): par = (1 1 1)
## 2651.2    (1.59e+02): par = (0.15468 -0.11928 2.576)
## 1690.9    (9.19e+01): par = (-0.015565 -0.14535 5.7642)
## 305.56    (3.46e+01): par = (-0.16284 0.18151 10.405)
## 32.728    (1.83e+00): par = (1.7376 3.463 12.847)
## Error in numericDeriv(form[[3L]], names(ind), env, central = nDcentral) :
##   Missing value or an infinity produced when evaluating the model
```

```
nlsfit0
```

```
## [1] "Error in numericDeriv(form[[3L]], names(ind), env, central = nDcentral) : \n  Missing value or a
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in numericDeriv(form[[3L]], names(ind), env, central = nDcentral): Missing value or an i
```

```
library(nlsr)
nlsrfit0 <- try(nlxb(y0 ~ aa * exp(-bb*tt) + cc, start=st, data=mydata, trace=FALSE))
nlsrfit0
```

```
## nlsr object: x
## residual sumsquares =  7.8564e-23  on  25 observations
##     after  36    Jacobian and  47 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## aa                   10     1.622e-10  6.165e+10  4.106e-224  -2.788e-14       615.5
## bb                 0.01     1.854e-13  5.393e+10   7.8e-223  -1.268e-12       3.198
## cc                    5     1.632e-10  3.065e+10  1.961e-217   7.638e-14    0.008214
```

We can approximate what `nls()` is doing by running a simple Gauss-Newton method one step at a time. It is (or was at the time of writing) easier to do this in functional form with explicit derivatives. Note that we test these functions!

```
trf <- function(par, data) {
    tt <- data[,"tt"]
    res <- par["aa"]*exp(-par["bb"]*tt) + par["cc"] - y0
}
print(trf(st, data=mydata))
```

```
##  [1] -13.533 -13.667 -13.655 -13.590 -13.506 -13.415 -13.323 -13.231 -13.139
## [10] -13.048 -12.958 -12.869 -12.781 -12.694 -12.607 -12.521 -12.437 -12.353
## [19] -12.270 -12.187 -12.106 -12.025 -11.945 -11.866 -11.788
```

```
trj <- function(par, data) {
    tt <- data[,"tt"]
    m <- dim(data)[1]
    JJ <- matrix(NA, nrow=m, ncol=3)
    JJ[,1] <- exp(-par["bb"]*tt)
    JJ[,2] <- - tt * par["aa"] * exp(-par["bb"]*tt)
    JJ[,3] <- 1
    JJ
}
Ja <- trj(st, data=mydata)
print(Ja)
```

```
##             [,1]        [,2] [,3]
##  [1,] 3.6788e-01 -3.6788e-01    1
##  [2,] 1.3534e-01 -2.7067e-01    1
##  [3,] 4.9787e-02 -1.4936e-01    1
##  [4,] 1.8316e-02 -7.3263e-02    1
##  [5,] 6.7379e-03 -3.3690e-02    1
##  [6,] 2.4788e-03 -1.4873e-02    1
##  [7,] 9.1188e-04 -6.3832e-03    1
##  [8,] 3.3546e-04 -2.6837e-03    1
##  [9,] 1.2341e-04 -1.1107e-03    1
## [10,] 4.5400e-05 -4.5400e-04    1
```

```
## [11,] 1.6702e-05 -1.8372e-04    1
## [12,] 6.1442e-06 -7.3731e-05    1
## [13,] 2.2603e-06 -2.9384e-05    1
## [14,] 8.3153e-07 -1.1641e-05    1
## [15,] 3.0590e-07 -4.5885e-06    1
## [16,] 1.1254e-07 -1.8006e-06    1
## [17,] 4.1399e-08 -7.0379e-07    1
## [18,] 1.5230e-08 -2.7414e-07    1
## [19,] 5.6028e-09 -1.0645e-07    1
## [20,] 2.0612e-09 -4.1223e-08    1
## [21,] 7.5826e-10 -1.5923e-08    1
## [22,] 2.7895e-10 -6.1368e-09    1
## [23,] 1.0262e-10 -2.3602e-09    1
## [24,] 3.7751e-11 -9.0603e-10    1
## [25,] 1.3888e-11 -3.4720e-10    1
```

```
library(numDeriv)
Jn <- jacobian(trf, st, data=mydata)
print(Jn)
```

```
##              [,1]        [,2] [,3]
##  [1,]  3.6788e-01 -3.6788e-01    1
##  [2,]  1.3534e-01 -2.7067e-01    1
##  [3,]  4.9787e-02 -1.4936e-01    1
##  [4,]  1.8316e-02 -7.3263e-02    1
##  [5,]  6.7379e-03 -3.3690e-02    1
##  [6,]  2.4788e-03 -1.4873e-02    1
##  [7,]  9.1188e-04 -6.3832e-03    1
##  [8,]  3.3546e-04 -2.6837e-03    1
##  [9,]  1.2341e-04 -1.1107e-03    1
## [10,]  4.5400e-05 -4.5400e-04    1
## [11,]  1.6702e-05 -1.8372e-04    1
## [12,]  6.1442e-06 -7.3731e-05    1
## [13,]  2.2603e-06 -2.9384e-05    1
## [14,]  8.3156e-07 -1.1641e-05    1
## [15,]  3.0599e-07 -4.5885e-06    1
## [16,]  1.1246e-07 -1.8006e-06    1
## [17,]  4.1441e-08 -7.0390e-07    1
## [18,]  1.5292e-08 -2.7413e-07    1
## [19,]  5.5106e-09 -1.0642e-07    1
## [20,]  2.0606e-09 -4.1195e-08    1
## [21,]  6.7789e-10 -1.5917e-08    1
## [22,]  2.8422e-10 -6.0949e-09    1
## [23,]  5.5252e-11 -2.3285e-09    1
## [24,] -1.5802e-11 -8.2053e-10    1
## [25,]  5.2006e-13 -3.5475e-10    1
```

```
print(max(abs(Jn-Ja)))
```

```
## [1] 1.0583e-10
```

```
ssf <- function(par, data){
   rr <- trf(par, data)
   ss <- crossprod(rr)
}
print(ssf(st, data=mydata))
```

```
##         [,1]
## [1,] 4092.5
```

```
library(numDeriv)
print(jacobian(trf, st, data=mydata))
```

```
##               [,1]        [,2] [,3]
##  [1,]  3.6788e-01 -3.6788e-01    1
##  [2,]  1.3534e-01 -2.7067e-01    1
##  [3,]  4.9787e-02 -1.4936e-01    1
##  [4,]  1.8316e-02 -7.3263e-02    1
##  [5,]  6.7379e-03 -3.3690e-02    1
##  [6,]  2.4788e-03 -1.4873e-02    1
##  [7,]  9.1188e-04 -6.3832e-03    1
##  [8,]  3.3546e-04 -2.6837e-03    1
##  [9,]  1.2341e-04 -1.1107e-03    1
## [10,]  4.5400e-05 -4.5400e-04    1
## [11,]  1.6702e-05 -1.8372e-04    1
## [12,]  6.1442e-06 -7.3731e-05    1
## [13,]  2.2603e-06 -2.9384e-05    1
## [14,]  8.3156e-07 -1.1641e-05    1
## [15,]  3.0599e-07 -4.5885e-06    1
## [16,]  1.1246e-07 -1.8006e-06    1
## [17,]  4.1441e-08 -7.0390e-07    1
## [18,]  1.5292e-08 -2.7413e-07    1
## [19,]  5.5106e-09 -1.0642e-07    1
## [20,]  2.0606e-09 -4.1195e-08    1
## [21,]  6.7789e-10 -1.5917e-08    1
## [22,]  2.8422e-10 -6.0949e-09    1
## [23,]  5.5252e-11 -2.3285e-09    1
## [24,] -1.5802e-11 -8.2053e-10    1
## [25,]  5.2006e-13 -3.5475e-10    1
```

The traditional way to implement a Gauss Newton method was to form the sum of squares and cross-products matrix at in traditional linear regression, using the Jacobian as the data matrix. The right hand side uses the inner product of the Jacobian with the negative residuals. This approach increases the condition number of the problem, and a more direct QR decomposition of the Jacobian is now the preferred approach. However, let us try both to ensure we are getting similar answers. First the QR approach.

```
gnjn <- function(start, resfn, jacfn = NULL, trace = FALSE,
        data=NULL, control=list(), ...){
# simplified Gauss Newton
   offset = 1e6 # for no change in parms
   stepred <- 0.5 # start with this as per nls()
   par <- start
   cat("starting parameters:")
   print(par)
   res <- resfn(par, data, ...)
   ssbest <- as.numeric(crossprod(res))
   cat("initial ss=",ssbest,"\n")
   par0 <- par
   kres <- 1
   kjac <- 0
   keepon <- TRUE
   while (keepon) {
      cat("kjac=",kjac,"  kres=",kres,"  SSbest now ", ssbest,"\n")
```

```
      print(par)
      JJ <- jacfn(par, data, ...)
      kjac <- kjac + 1
      QJ <- qr(JJ)
      delta <- qr.coef(QJ, -res)
      ss <- ssbest + offset*offset # force evaluation
      step <- 1.0
      if (as.numeric(max(par0+delta)+offset) != as.numeric(max(par0+offset)) ) {
         while (ss > ssbest) {
           par <- par0+delta * step
           res <- resfn(par, data, ...)
           ss <- as.numeric(crossprod(res))
           kres <- kres + 1
##           cat("step =", step,"  ss=",ss,"\n")
##           tmp <- readline("continue")
           if (ss > ssbest) {
              step <- step * stepred
           } else {
              par0 <- par
              ssbest <- ss
           }
         } # end inner loop
         if (kjac >= 4)  {
            keepon = FALSE
            cat("artificial stop at kjac=4 -- we only want to check output")
         }
      } else { keepon <- FALSE # done }
   } # end main iteration
} # seems to need this

} # end gnjne

fitgnjn0 <- gnjn(st, trf, trj, data=mydata)
## Another way
#- set lamda = 0 in nlxb, fix laminc, lamdec
library(nlsr)
nlx00 <- try(nlxb(y0 ~ aa * exp(-bb*tt) + cc, start=st, data=mydata, trace=TRUE,
                    control=list(lamda=0, laminc=0, lamdec=0, watch=TRUE)))
nlx00
```

The first iteration more or less matches the `nls()` result. The problem is quite ill-conditioned, and `nls()` is using a numerical approximation to the Jacobian, so deviations of this magnitude from the iterations are not unexpected.

Let us test with a traditional Gauss-Newton.

```
gnjn2 <- function(start, resfn, jacfn = NULL, trace = FALSE,
        data=NULL, control=list(), ...){
# simplified Gauss Newton
   offset = 1e6 # for no change in parms
   stepred <- 0.5 # start with this as per nls()
   par <- start
   cat("starting parameters:")
   print(par)
   res <- resfn(par, data, ...)
```

```
   ssbest <- as.numeric(crossprod(res))
   cat("initial ss=",ssbest,"\n")
   kres <- 1
   kjac <- 0
   par0 <- par
   keepon <- TRUE
   while (keepon) {
      cat("kres=",kres,"  kjac=",kjac,"   SSbest now ", ssbest,"\n")
      print(par)
      JJ <- jacfn(par, data, ...)
      kjac <- kjac + 1
      JTJ <- crossprod (JJ)
      JTr <- crossprod (JJ, res)
      delta <- - as.vector(solve(JTJ, JTr))
      ss <- ssbest + offset*offset # force evaluation
      step <- 1.0
      if (as.numeric(max(par0+delta)+offset) != as.numeric(max(par0+offset)) ) {
         while (ss > ssbest) {
           par <- par0+delta * step
           res <- resfn(par, data, ...)
           ss <- as.numeric(crossprod(res))
           kres <- kres + 1
##           cat("step =", step,"  ss=",ss,"  best is",ssbest,"\n")
##           tmp <- readline("continue")
           if (ss > ssbest) {
              step <- step * stepred
           } else {
              par0 <- par
              ssbest <- ss
           }
         } # end inner loop
         if (kjac >= 4)  {
            keepon = FALSE
            cat("artificial stop at kjac=4 -- we only want to check output")
         }
      } else { keepon <- FALSE # done }
   } # end main iteration
} # seems to need this
} # end gnjn2

fitgnjn20 <- gnjn2(st, trf, trj, data=mydata)
```

Implementing the Marquardt stabilization in QR form

?? to add

Implementing a relative offset convergence test

Nonlinear equations

Solution of sets of nonlinear equations is generally NOT a problem that is commonly required for statisticians or data analysts. My experience is that the occasions where it does arise are when workers try to solve the first order conditions for optimality of a function, rather than try to optimize the function. If this function is a sum of squares, then we have a nonlinear least squares problem, and generally such problems are best

approached my methods of the type discussed in this article.

Conversely, since our problem is, using the notation above, equivalent to

r(x) = 0

the solution of a nonlinear least squares problem for which the final sum of squares is zero provides a solution to the nonlinear equations. In my experience this is a valid approach to the nonlinear equations problem, especially if there is concern that a solution may not exist. Note that there are methods for nonlinear equations, some of which (??ref) are available in R packages.

Function versus expression approach

In `nlsr`, we provide for modelling via R functions that generate the residuals and Jacobian, and we call the nonlinear least squares minimizer through `nlfb`. Alternatively, we can do the modelling via the `nlxb` function that takes a model expression as an argument and converts it to the functional form. The actual least squares minimization is then carried out by a common routine (`nlfb`). Generally `nlxb` is a lot less programming work, but it is also more limited, as we can only handle single line expressions, and some expressions may not be differentiable within the package, even if there are mathematically feasible ways to compute them analytically.

Providing extra data to expressions

Almost all statistical functions have exogenous data that is needed to compute residuals or likelihoods and is not dependent on the model parameters. (This section starts from Notes140806.)

model2rjfun does NOT have ... args.

Should it have? i.e., a problem where we are fitting a set of time series, 1 for each plant/animal, with some sort of start parameter for each that is NOT estimated (e.g., pH of soil, some index of health).

Difficulty in such a problem is that the residuals are then a matrix, and the nlfb rather than nlxb is a better approach. However, fitting 1 series would still need this data, and example nlsrtryextraparms.txt shows that the extra parm (ms in this case) needs to be in the user's globalenv.

```
rm(list=ls())
require(nlsr)
# want to have data AND extra parameters (NOT to be estimated)
traceval  <-  TRUE  # traceval set TRUE to debug or give full history
# Data for Hobbs problem
ydat  <-  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) # for testing
tdat  <-  seq_along(ydat) # for testing
# A simple starting vector -- must have named parameters for nlxb, nls, wrapnlsr.
start1  <-  c(b1=1, b2=1, b3=1)
startf1  <-  c(b1=1, b2=1, b3=.1)
eunsc  <-   y ~ b1/(1+b2*exp(-b3*tt))
cat("LOCAL DATA IN DATA FRAMES\n")
```

```
## LOCAL DATA IN DATA FRAMES
```

```
weeddata1  <-  data.frame(y=ydat, tt=tdat)
cat("weeddata contains the original data\n")
```

```
## weeddata contains the original data
```

```
ms <- 2 # define the external parameter here
cat("wdata scales y by ms =",ms,"\n")
```

```
## wdata scales y by ms = 2
```

```
wdata <- data.frame(y=ydat/ms, tt=tdat)
wdata
```

```
##          y tt
## 1   2.6540  1
## 2   3.6200  2
## 3   4.8190  3
## 4   6.4330  4
## 5   8.5345  5
## 6  11.5960  6
## 7  15.7215  7
## 8  19.2790  8
## 9  25.0780  9
## 10 31.4740 10
## 11 37.9975 11
## 12 45.9860 12
```

```
cat("estimate the UNSCALED model with SCALED data\n")
```

```
## estimate the UNSCALED model with SCALED data
```

```
anlxbs  <-  try(nlxb(eunsc, start=start1, trace=traceval, data=wdata))
```

```
## formula: y ~ b1/(1 + b2 * exp(-b3 * tt))
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## $watch
## [1] FALSE
##
## $phi
## [1] 1
##
## $lamda
## [1] 1e-04
##
## $offset
## [1] 100
##
## $laminc
## [1] 10
##
## $lamdec
## [1] 4
##
## $femax
## [1] 10000
##
## $jemax
## [1] 5000
##
## $rofftest
## [1] TRUE
##
## $smallsstest
## [1] TRUE
```

```
##
## Finished masks check
## datvar:[1] "y"  "tt"
## Data variable  y : [1]  2.6540  3.6200  4.8190  6.4330  8.5345 11.5960 15.7215 19.2790 25.0780
## [10] 31.4740 37.9975 45.9860
## Data variable  tt : [1]  1  2  3  4  5  6  7  8  9 10 11 12
## trjfn:
## function (prm)
## {
##     if (is.null(names(prm)))
##         names(prm) <- names(pvec)
##     localdata <- list2env(as.list(prm), parent = data)
##     eval(residexpr, envir = localdata)
## }
## <bytecode: 0x55c43b156848>
## <environment: 0x55c43cb95120>
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Start:lamda: 1e-04  SS= 5676.9  at  b1 = 1  b2 = 1  b3 = 1  1 / 0
## lamda: 0.001  SS= 6088.9  at  b1 = 25.443  b2 = -119.86  b3 = -185.95  2 / 1
## lamda: 0.01  SS= 6088.9  at  b1 = 25.092  b2 = -94.85  b3 = -166.61  3 / 1
## lamda: 0.1  SS= 6088.9  at  b1 = 23.492  b2 = -12.164  b3 = -96.948  4 / 1
## lamda: 1  SS= 6088.9  at  b1 = 19.105  b2 = 15.514  b3 = -31.768  5 / 1
## lamda: 10  SS= 6078  at  b1 = 9.6639  b2 = 2.3414  b3 = -1.0745  6 / 1
## <<lamda: 4  SS= 5096.5  at  b1 = 2.5089  b2 = 0.94659  b3 = 1.1409  7 / 1
## <<lamda: 1.6  SS= 4086.7  at  b1 = 5.5395  b2 = 1.1277  b3 = 0.98661  8 / 2
## lamda: 16  SS= 5979.2  at  b1 = 10.709  b2 = 2.4361  b3 = -0.35798  9 / 3
## <<lamda: 6.4  SS= 3871.1  at  b1 = 6.2757  b2 = 1.195  b3 = 0.95044  10 / 3
## <<lamda: 2.56  SS= 3427.1  at  b1 = 7.914  b2 = 1.4621  b3 = 0.76952  11 / 4
## <<lamda: 1.024  SS= 2714  at  b1 = 11.243  b2 = 2.2923  b3 = 0.41244  12 / 5
## <<lamda: 0.4096  SS= 1624.3  at  b1 = 17.581  b2 = 3.9784  b3 = 0.51333  13 / 6
## <<lamda: 0.16384  SS= 1596  at  b1 = 25.778  b2 = 7.8443  b3 = 0.25361  14 / 7
## <<lamda: 0.065536  SS= 389.99  at  b1 = 36.58  b2 = 11.353  b3 = 0.40144  15 / 8
## <<lamda: 0.026214  SS= 227.36  at  b1 = 47.307  b2 = 19.475  b3 = 0.32531  16 / 9
## <<lamda: 0.010486  SS= 30.768  at  b1 = 61.38  b2 = 30.701  b3 = 0.35494  17 / 10
## <<lamda: 0.0041943  SS= 7.5356  at  b1 = 71.376  b2 = 39.434  b3 = 0.3436  18 / 11
## <<lamda: 0.0016777  SS= 2.4081  at  b1 = 80.801  b2 = 44.729  b3 = 0.33443  19 / 12
## <<lamda: 0.00067109  SS= 1.1886  at  b1 = 88.884  b2 = 47.007  b3 = 0.32377  20 / 13
## <<lamda: 0.00026844  SS= 0.72638  at  b1 = 94.691  b2 = 48.297  b3 = 0.31699  21 / 14
## <<lamda: 0.00010737  SS= 0.6497  at  b1 = 97.378  b2 = 48.922  b3 = 0.31428  22 / 15
## <<lamda: 4.295e-05  SS= 0.64684  at  b1 = 98.022  b2 = 49.075  b3 = 0.31364  23 / 16
## <<lamda: 1.718e-05  SS= 0.64682  at  b1 = 98.09  b2 = 49.091  b3 = 0.31357  24 / 17
## <<lamda: 6.8719e-06  SS= 0.64682  at  b1 = 98.093  b2 = 49.092  b3 = 0.31357  25 / 18
## <<lamda: 2.7488e-06  SS= 0.64682  at  b1 = 98.093  b2 = 49.092  b3 = 0.31357  26 / 19
## <<lamda: 1.0995e-06  SS= 0.64682  at  b1 = 98.093  b2 = 49.092  b3 = 0.31357  27 / 20
```

```
print(anlxbs)
```

```
## nlsr object: x
## residual sumsquares =  0.64682  on  12 observations
##     after  20    Jacobian and  27 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1              98.0931         5.653       17.35  3.167e-08  -4.548e-12       505.4
## b2              49.0916         1.688       29.08  3.284e-10  -1.619e-12      0.2344
```

```
## b3               0.31357      0.006863      45.69  5.768e-12   7.218e-10     0.04632
```

```
escal <-  y ~ ms*b1/(1+b2*exp(-b3*tt))
cat("estimate the SCALED model with scaling provided in the call (ms=0.5)\n")
```

```
## estimate the SCALED model with scaling provided in the call (ms=0.5)
```

```
anlxbh  <-  try(nlxb(escal, start=start1, trace=traceval, data=weeddata1, ms=0.5))
```

```
## Error in nlxb(escal, start = start1, trace = traceval, data = weeddata1,  :
##   unused argument (ms = 0.5)
```

```
print(anlxbh)
```

```
## [1] "Error in nlxb(escal, start = start1, trace = traceval, data = weeddata1,  : \n  unused argument
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in nlxb(escal, start = start1, trace = traceval, data = weeddata1,     ms = 0.5): unuse
```

```
cat("\n scaling is now using the globally defined value of ms=",ms,"\n")
```

```
##
##  scaling is now using the globally defined value of ms= 2
```

```
anlxb1a  <-  try(nlxb(escal, start=start1, trace=traceval, data=weeddata1))
```

```
## formula: y ~ ms * b1/(1 + b2 * exp(-b3 * tt))
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## $watch
## [1] FALSE
##
## $phi
## [1] 1
##
## $lamda
## [1] 1e-04
##
## $offset
## [1] 100
##
## $laminc
## [1] 10
##
## $lamdec
## [1] 4
##
## $femax
## [1] 10000
##
## $jemax
## [1] 5000
##
## $rofftest
## [1] TRUE
##
## $smallsstest
```

```
## [1] TRUE
##
## Finished masks check
## datvar:[1] "y"  "ms" "tt"
## Data variable  y : [1]  5.308  7.240  9.638 12.866 17.069 23.192 31.443 38.558 50.156 62.948
## [11] 75.995 91.972
## Data variable  ms :[1] 2
## Data variable  tt : [1]  1  2  3  4  5  6  7  8  9 10 11 12
## trjfn:
## function (prm)
## {
##     if (is.null(names(prm)))
##         names(prm) <- names(pvec)
##     localdata <- list2env(as.list(prm), parent = data)
##     eval(residexpr, envir = localdata)
## }
## <bytecode: 0x55c43b156848>
## <environment: 0x55c43a479378>
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Start:lamda: 1e-04  SS= 22708  at  b1 = 1  b2 = 1  b3 = 1  1 / 0
## lamda: 0.001  SS= 24356  at  b1 = 25.472  b2 = -122.14  b3 = -187.69  2 / 1
## lamda: 0.01  SS= 24356  at  b1 = 25.327  b2 = -113.2  b3 = -180.68  3 / 1
## lamda: 0.1  SS= 24356  at  b1 = 24.283  b2 = -57.588  b3 = -135.68  4 / 1
## lamda: 1  SS= 24356  at  b1 = 20.254  b2 = 14.461  b3 = -54.127  5 / 1
## lamda: 10  SS= 24355  at  b1 = 10.077  b2 = 4.8657  b3 = -4.5699  6 / 1
## <<lamda: 4  SS= 20252  at  b1 = 2.5977  b2 = 0.83113  b3 = 1.4117  7 / 1
## <<lamda: 1.6  SS= 16143  at  b1 = 5.7092  b2 = 1.3529  b3 = 0.86529  8 / 2
## lamda: 16  SS= 22536  at  b1 = 11.269  b2 = 3.0589  b3 = -0.12239  9 / 3
## <<lamda: 6.4  SS= 15214  at  b1 = 6.5093  b2 = 1.428  b3 = 0.86138  10 / 3
## <<lamda: 2.56  SS= 13336  at  b1 = 8.2714  b2 = 1.7659  b3 = 0.74306  11 / 4
## <<lamda: 1.024  SS= 10290  at  b1 = 11.801  b2 = 2.7987  b3 = 0.46561  12 / 5
## <<lamda: 0.4096  SS= 5999.3  at  b1 = 18.477  b2 = 4.9891  b3 = 0.46298  13 / 6
## <<lamda: 0.16384  SS= 3360.8  at  b1 = 27.609  b2 = 9.7076  b3 = 0.35379  14 / 7
## <<lamda: 0.065536  SS= 872.75  at  b1 = 40.302  b2 = 16.934  b3 = 0.38655  15 / 8
## <<lamda: 0.026214  SS= 308.99  at  b1 = 51.624  b2 = 26.628  b3 = 0.36281  16 / 9
## <<lamda: 0.010486  SS= 64.882  at  b1 = 63.651  b2 = 36.557  b3 = 0.35778  17 / 10
## <<lamda: 0.0041943  SS= 19.992  at  b1 = 73.76  b2 = 43.123  b3 = 0.3463  18 / 11
## <<lamda: 0.0016777  SS= 8.8066  at  b1 = 83.053  b2 = 46.013  b3 = 0.33222  19 / 12
## <<lamda: 0.00067109  SS= 4.1734  at  b1 = 91.086  b2 = 47.556  b3 = 0.32103  20 / 13
## <<lamda: 0.00026844  SS= 2.7218  at  b1 = 96.072  b2 = 48.622  b3 = 0.31554  21 / 14
## <<lamda: 0.00010737  SS= 2.5891  at  b1 = 97.8  b2 = 49.023  b3 = 0.31386  22 / 15
## <<lamda: 4.295e-05  SS= 2.5873  at  b1 = 98.074  b2 = 49.087  b3 = 0.31359  23 / 16
## <<lamda: 1.718e-05  SS= 2.5873  at  b1 = 98.093  b2 = 49.091  b3 = 0.31357  24 / 17
## <<lamda: 6.8719e-06  SS= 2.5873  at  b1 = 98.093  b2 = 49.092  b3 = 0.31357  25 / 18
## <<lamda: 2.7488e-06  SS= 2.5873  at  b1 = 98.093  b2 = 49.092  b3 = 0.31357  26 / 19
## lamda: 2.7488e-05  SS= 2.5873  at  b1 = 98.093  b2 = 49.092  b3 = 0.31357  27 / 20
```

```
print(anlxb1a)
```

```
## nlsr object: x
## residual sumsquares =  2.5873  on  12 observations
##     after  20    Jacobian and  27 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
```

```
## b1               98.0931         5.653      17.35  3.167e-08  -2.405e-10        1011
## b2               49.0916         1.688      29.08  3.284e-10  -2.315e-09      0.4687
## b3               0.31357      0.006863      45.69  5.768e-12   2.754e-07     0.09263
```

```
ms <- 1
cat("\n scaling is now using the globally re-defined value of ms=",ms,"\n")
```

```
##
##  scaling is now using the globally re-defined value of ms= 1
```

```
anlxb1b  <-  try(nlxb(escal, start=start1, trace=traceval, data=weeddata1))
```

```
## formula: y ~ ms * b1/(1 + b2 * exp(-b3 * tt))
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## $watch
## [1] FALSE
##
## $phi
## [1] 1
##
## $lamda
## [1] 1e-04
##
## $offset
## [1] 100
##
## $laminc
## [1] 10
##
## $lamdec
## [1] 4
##
## $femax
## [1] 10000
##
## $jemax
## [1] 5000
##
## $rofftest
## [1] TRUE
##
## $smallsstest
## [1] TRUE
##
## Finished masks check
## datvar:[1] "y"  "ms" "tt"
## Data variable  y : [1]  5.308  7.240  9.638 12.866 17.069 23.192 31.443 38.558 50.156 62.948
## [11] 75.995 91.972
## Data variable  ms :[1] 1
## Data variable  tt : [1]  1  2  3  4  5  6  7  8  9 10 11 12
## trjfn:
## function (prm)
## {
##     if (is.null(names(prm)))
```

```
##         names(prm) <- names(pvec)
##     localdata <- list2env(as.list(prm), parent = data)
##     eval(residexpr, envir = localdata)
## }
## <bytecode: 0x55c43b156848>
## <environment: 0x55c43df00be0>
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Start:lamda: 1e-04  SS= 23521  at  b1 = 1  b2 = 1  b3 = 1  1 / 0
## lamda: 0.001  SS= 24356  at  b1 = 50.886  b2 = -240.72  b3 = -372.91  2 / 1
## lamda: 0.01  SS= 24356  at  b1 = 50.183  b2 = -190.69  b3 = -334.2  3 / 1
## lamda: 0.1  SS= 24356  at  b1 = 46.97  b2 = -25.309  b3 = -194.81  4 / 1
## lamda: 1  SS= 24356  at  b1 = 38.096  b2 = 29.924  b3 = -64.262  5 / 1
## lamda: 10  SS= 24353  at  b1 = 18.798  b2 = 3.551  b3 = -2.9011  6 / 1
## <<lamda: 4  SS= 21065  at  b1 = 4.1015  b2 = 0.86688  b3 = 1.3297  7 / 1
## <<lamda: 1.6  SS= 16852  at  b1 = 10.248  b2 = 1.2302  b3 = 1.0044  8 / 2
## lamda: 16  SS= 24078  at  b1 = 20.944  b2 = 2.9473  b3 = -0.40959  9 / 3
## <<lamda: 6.4  SS= 15934  at  b1 = 11.771  b2 = 1.3084  b3 = 0.98087  10 / 3
## <<lamda: 2.56  SS= 14050  at  b1 = 15.152  b2 = 1.6468  b3 = 0.80462  11 / 4
## <<lamda: 1.024  SS= 10985  at  b1 = 22.005  b2 = 2.6632  b3 = 0.45111  12 / 5
## <<lamda: 0.4096  SS= 6427.1  at  b1 = 35.128  b2 = 4.7135  b3 = 0.50974  13 / 6
## <<lamda: 0.16384  SS= 4725  at  b1 = 52.285  b2 = 9.2948  b3 = 0.31348  14 / 7
## <<lamda: 0.065536  SS= 1132.2  at  b1 = 76.446  b2 = 15.142  b3 = 0.41095  15 / 8
## <<lamda: 0.026214  SS= 518.76  at  b1 = 96.924  b2 = 24.422  b3 = 0.35816  16 / 9
## <<lamda: 0.010486  SS= 79.464  at  b1 = 122.18  b2 = 35.262  b3 = 0.36484  17 / 10
## <<lamda: 0.0041943  SS= 26.387  at  b1 = 141.55  b2 = 42.387  b3 = 0.35215  18 / 11
## <<lamda: 0.0016777  SS= 11.339  at  b1 = 160.02  b2 = 45.519  b3 = 0.33738  19 / 12
## <<lamda: 0.00067109  SS= 5.2767  at  b1 = 176.92  b2 = 47.037  b3 = 0.32443  20 / 13
## <<lamda: 0.00026844  SS= 2.9656  at  b1 = 189.12  b2 = 48.279  b3 = 0.31712  21 / 14
## <<lamda: 0.00010737  SS= 2.6003  at  b1 = 194.72  b2 = 48.92  b3 = 0.31429  22 / 15
## <<lamda: 4.295e-05  SS= 2.5873  at  b1 = 196.04  b2 = 49.075  b3 = 0.31364  23 / 16
## <<lamda: 1.718e-05  SS= 2.5873  at  b1 = 196.18  b2 = 49.091  b3 = 0.31357  24 / 17
## <<lamda: 6.8719e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  25 / 18
## <<lamda: 2.7488e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  26 / 19
## <<lamda: 1.0995e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  27 / 20
```

```
print(anlxb1b)
```

```
## nlsr object: x
## residual sumsquares =  2.5873  on  12 observations
##     after  20    Jacobian and  27 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1              196.186         11.31       17.35  3.167e-08  -8.803e-12        1011
## b2              49.0916         1.688       29.08  3.284e-10  -4.061e-12      0.4605
## b3              0.31357      0.006863       45.69  5.768e-12   2.832e-09     0.04714
```

Examples of use

Nonlinear least squares with expressions

We use the Hobbs Weeds problem (Nash, 1979 and Nash, 2014). Note that nls() fails from start1.

```
require(nlsr)
traceval <- FALSE
# Data for Hobbs problem
```

```
ydat  <-  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) # for testing
tdat  <-  seq_along(ydat) # for testing

# A simple starting vector -- must have named parameters for nlxb, nls, wrapnlsr.
start1  <-  c(b1=1, b2=1, b3=1)

eunsc  <-   y ~ b1/(1+b2*exp(-b3*tt))

cat("LOCAL DATA IN DATA FRAMES\n")
```

```
## LOCAL DATA IN DATA FRAMES
```

```
weeddata1  <-  data.frame(y=ydat, tt=tdat)
weeddata2  <-  data.frame(y=1.5*ydat, tt=tdat)

anlxb1  <-  try(nlxb(eunsc, start=start1, trace=TRUE, data=weeddata1,
                     control=list(watch=FALSE)))
```

```
## formula: y ~ b1/(1 + b2 * exp(-b3 * tt))
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## $watch
## [1] FALSE
##
## $phi
## [1] 1
##
## $lamda
## [1] 1e-04
##
## $offset
## [1] 100
##
## $laminc
## [1] 10
##
## $lamdec
## [1] 4
##
## $femax
## [1] 10000
##
## $jemax
## [1] 5000
##
## $rofftest
## [1] TRUE
##
## $smallsstest
## [1] TRUE
##
## Finished masks check
## datvar:[1] "y"  "tt"
## Data variable  y : [1]  5.308  7.240  9.638 12.866 17.069 23.192 31.443 38.558 50.156 62.948
```

```
## [11] 75.995 91.972
## Data variable  tt : [1]  1  2  3  4  5  6  7  8  9 10 11 12
## trjfn:
## function (prm)
## {
##     if (is.null(names(prm)))
##         names(prm) <- names(pvec)
##     localdata <- list2env(as.list(prm), parent = data)
##     eval(residexpr, envir = localdata)
## }
## <bytecode: 0x55c43b156848>
## <environment: 0x55c43c999a78>
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Start:lamda: 1e-04  SS= 23521  at  b1 = 1  b2 = 1  b3 = 1  1 / 0
## lamda: 0.001  SS= 24356  at  b1 = 50.886  b2 = -240.72  b3 = -372.91  2 / 1
## lamda: 0.01  SS= 24356  at  b1 = 50.183  b2 = -190.69  b3 = -334.2  3 / 1
## lamda: 0.1  SS= 24356  at  b1 = 46.97  b2 = -25.309  b3 = -194.81  4 / 1
## lamda: 1  SS= 24356  at  b1 = 38.096  b2 = 29.924  b3 = -64.262  5 / 1
## lamda: 10  SS= 24353  at  b1 = 18.798  b2 = 3.551  b3 = -2.9011  6 / 1
## <<lamda: 4  SS= 21065  at  b1 = 4.1015  b2 = 0.86688  b3 = 1.3297  7 / 1
## <<lamda: 1.6  SS= 16852  at  b1 = 10.248  b2 = 1.2302  b3 = 1.0044  8 / 2
## lamda: 16  SS= 24078  at  b1 = 20.944  b2 = 2.9473  b3 = -0.40959  9 / 3
## <<lamda: 6.4  SS= 15934  at  b1 = 11.771  b2 = 1.3084  b3 = 0.98087  10 / 3
## <<lamda: 2.56  SS= 14050  at  b1 = 15.152  b2 = 1.6468  b3 = 0.80462  11 / 4
## <<lamda: 1.024  SS= 10985  at  b1 = 22.005  b2 = 2.6632  b3 = 0.45111  12 / 5
## <<lamda: 0.4096  SS= 6427.1  at  b1 = 35.128  b2 = 4.7135  b3 = 0.50974  13 / 6
## <<lamda: 0.16384  SS= 4725  at  b1 = 52.285  b2 = 9.2948  b3 = 0.31348  14 / 7
## <<lamda: 0.065536  SS= 1132.2  at  b1 = 76.446  b2 = 15.142  b3 = 0.41095  15 / 8
## <<lamda: 0.026214  SS= 518.76  at  b1 = 96.924  b2 = 24.422  b3 = 0.35816  16 / 9
## <<lamda: 0.010486  SS= 79.464  at  b1 = 122.18  b2 = 35.262  b3 = 0.36484  17 / 10
## <<lamda: 0.0041943  SS= 26.387  at  b1 = 141.55  b2 = 42.387  b3 = 0.35215  18 / 11
## <<lamda: 0.0016777  SS= 11.339  at  b1 = 160.02  b2 = 45.519  b3 = 0.33738  19 / 12
## <<lamda: 0.00067109  SS= 5.2767  at  b1 = 176.92  b2 = 47.037  b3 = 0.32443  20 / 13
## <<lamda: 0.00026844  SS= 2.9656  at  b1 = 189.12  b2 = 48.279  b3 = 0.31712  21 / 14
## <<lamda: 0.00010737  SS= 2.6003  at  b1 = 194.72  b2 = 48.92  b3 = 0.31429  22 / 15
## <<lamda: 4.295e-05  SS= 2.5873  at  b1 = 196.04  b2 = 49.075  b3 = 0.31364  23 / 16
## <<lamda: 1.718e-05  SS= 2.5873  at  b1 = 196.18  b2 = 49.091  b3 = 0.31357  24 / 17
## <<lamda: 6.8719e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  25 / 18
## <<lamda: 2.7488e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  26 / 19
## <<lamda: 1.0995e-06  SS= 2.5873  at  b1 = 196.19  b2 = 49.092  b3 = 0.31357  27 / 20
```

```
print(anlxb1)
```

```
## nlsr object: x
## residual sumsquares =  2.5873  on  12 observations
##     after  20    Jacobian and  27 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               196.186         11.31       17.35  3.167e-08  -8.803e-12        1011
## b2               49.0916         1.688       29.08  3.284e-10  -4.061e-12      0.4605
## b3               0.31357      0.006863       45.69  5.768e-12   2.832e-09     0.04714
```

```
anlsb1 <-  try(nls(eunsc, start=start1, trace=TRUE, data=weeddata1))
```

```
## 23521.    (2.08e+00): par = (1 1 1)
## Error in nls(eunsc, start = start1, trace = TRUE, data = weeddata1) :
##   singular gradient
```

```
print(anlsb1)
```

```
## [1] "Error in nls(eunsc, start = start1, trace = TRUE, data = weeddata1) : \n  singular gradient\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in nls(eunsc, start = start1, trace = TRUE, data = weeddata1): singular gradient>
```

A different start causes nlxb to return a large sum of squares. Note that nls() again fails.

```
startf1  <-  c(b1=1, b2=1, b3=.1)
anlsf1 <-  try(nls(eunsc, start=startf1, trace=TRUE, data=weeddata1))
```

```
## 23756.    (4.65e+01): par = (1 1 0.1)
## Error in nls(eunsc, start = startf1, trace = TRUE, data = weeddata1) :
##   singular gradient
```

```
print(anlsf1)
```

```
## [1] "Error in nls(eunsc, start = startf1, trace = TRUE, data = weeddata1) : \n  singular gradient\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in nls(eunsc, start = startf1, trace = TRUE, data = weeddata1): singular gradient>
```

```
library(nlsr)
anlxf1  <-  try(nlxb(eunsc, start=startf1, trace=TRUE, data=weeddata1,
                     control=list(watch=FALSE)))
```

```
## formula: y ~ b1/(1 + b2 * exp(-b3 * tt))
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## $watch
## [1] FALSE
##
## $phi
## [1] 1
##
## $lamda
## [1] 1e-04
##
## $offset
## [1] 100
##
## $laminc
## [1] 10
##
## $lamdec
## [1] 4
##
## $femax
```

```
## [1] 10000
##
## $jemax
## [1] 5000
##
## $rofftest
## [1] TRUE
##
## $smallsstest
## [1] TRUE
##
## Finished masks check
## datvar:[1] "y"  "tt"
## Data variable  y : [1]  5.308  7.240  9.638 12.866 17.069 23.192 31.443 38.558 50.156 62.948
## [11] 75.995 91.972
## Data variable  tt : [1]  1  2  3  4  5  6  7  8  9 10 11 12
## trjfn:
## function (prm)
## {
##     if (is.null(names(prm)))
##         names(prm) <- names(pvec)
##     localdata <- list2env(as.list(prm), parent = data)
##     eval(residexpr, envir = localdata)
## }
## <bytecode: 0x55c43b156848>
## <environment: 0x55c43dda1c48>
## no weights
## lower:[1] -Inf -Inf -Inf
## upper:[1] Inf Inf Inf
## Start:lamda: 1e-04  SS= 23756  at  b1 = 1  b2 = 1  b3 = 0.1  1 / 0
## lamda: 0.001  SS= 24356  at  b1 = 416.72  b2 = 821.43  b3 = -40.851  2 / 1
## lamda: 0.01  SS= 196562  at  b1 = 162.93  b2 = 368.16  b3 = 7.5932  3 / 1
## <<lamda: 0.004  SS= 10597  at  b1 = 24.764  b2 = 108.11  b3 = 31.926  4 / 1
## <<lamda: 0.0016  SS= 9205.5  at  b1 = 35.486  b2 = 108.11  b3 = 31.926  5 / 2
## <<lamda: 0.00064  SS= 9205.4  at  b1 = 35.532  b2 = 108.11  b3 = 31.926  6 / 3
## <<lamda: 0.000256  SS= 9205.4  at  b1 = 35.532  b2 = 108.11  b3 = 31.926  7 / 4
## lamda: 0.00256  SS= 9205.4  at  b1 = 35.532  b2 = 108.11  b3 = 31.926  8 / 5
```

```
print(anlxf1)
```

```
## nlsr object: x
## residual sumsquares =  9205.4  on  12 observations
##     after  5    Jacobian and  8 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1              35.5321            NA         NA         NA  -6.684e-07       3.464
## b2              108.109            NA         NA         NA  -1.464e-11   5.015e-11
## b3              31.9262            NA         NA         NA   1.583e-09   6.143e-27
```

```
# anlxb2  <-  try(nlxb(eunsc, start=start1, trace=FALSE, data=weeddata2))
# print(anlxb2)
```

We can discover quickly the difficulty here by computing the Jacobian at this "solution" and checking its singular values.

```
cf1 <- coef(anlxf1)
print(cf1)
```

```
##      b1      b2      b3
##  35.532 108.109  31.926
## attr(,"pkgname")
## [1] "nlsr"
```

```
jf1 <- anlxf1$jacobian
svals <- svd(jf1)$d
print(svals)
```

```
## [1] 3.4641e+00 5.0147e-11 6.1432e-27
```

Here we see that the Jacobian is only rank 1, even though there are 3 coefficients. It is therefore not surprising that our nonlinear least squares program has concluded we are unable to make further progress.

Nonlinear least squares with functions

We can run the same example as above using **R** functions rather than expressions, but now we need to have a gradient function as well as one to compute residuals. **nlsr** has tools to create these functions from expressions, as we shall see here. First we again set up the data and load the package.

```
require(nlsr)
traceval <- FALSE
# Data for Hobbs problem
ydat  <-  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) # for testing
tdat  <-  seq_along(ydat) # for testing

# A simple starting vector -- must have named parameters for nlxb, nls, wrapnlsr.
start1  <-  c(b1=1, b2=1, b3=1)

eunsc  <-   y ~ b1/(1+b2*exp(-b3*tt))

cat("LOCAL DATA IN DATA FRAMES\n")
```

```
## LOCAL DATA IN DATA FRAMES
```

```
weeddata1  <-  data.frame(y=ydat, tt=tdat)


weedrj <- model2rjfun(modelformula=eunsc, pvec=start1, data=weeddata1)
weedrj
```

```
## function (prm)
## {
##     if (is.null(names(prm)))
##         names(prm) <- names(pvec)
##     localdata <- list2env(as.list(prm), parent = data)
##     eval(residexpr, envir = localdata)
## }
## <bytecode: 0x55c43b156848>
## <environment: 0x55c43d0d5100>
```

```
rjfundoc(weedrj) # Note how useful this is to report status
```

```
## FUNCTION weedrj
## Formula: y ~ b1/(1 + b2 * exp(-b3 * tt))
## Code:        expression({
##     .expr3 <- exp(-b3 * tt)
```

```
##     .expr5 <- 1 + b2 * .expr3
##     .expr10 <- .expr5^2
##     .value <- b1/.expr5 - y
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("b1",
##         "b2", "b3")))
##     .grad[, "b1"] <- 1/.expr5
##     .grad[, "b2"] <- -(b1 * .expr3/.expr10)
##     .grad[, "b3"] <- b1 * (b2 * (.expr3 * tt))/.expr10
##     attr(.value, "gradient") <- .grad
##     .value
## })
## Parameters:   b1, b2, b3
## Data:         y, tt
##
## VALUES
## Observations:     12
## Parameters:
## b1 b2 b3
##  1  1  1
## Data (length 12):
##         y tt
## 1   5.308  1
## 2   7.240  2
## 3   9.638  3
## 4  12.866  4
## 5  17.069  5
## 6  23.192  6
## 7  31.443  7
## 8  38.558  8
## 9  50.156  9
## 10 62.948 10
## 11 75.995 11
## 12 91.972 12
```

check modelexpr() works with an ssgrfun ??

test model2rjfun vs model2rjfunx ??

Why is resss difficult to use in optimization?

```
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 <- seq_along(y) # for testing
mydata <- data.frame(y = y, tt = tt)
f <- y ~ 100*b1/(1 + 10*b2 * exp(-0.1 * b3 * tt))
p <- c(b1 = 1, b2 = 1, b3 = 1)
rjfn <- model2rjfun(f, p, data = mydata)
rjfn(p)
```

```
##  [1]   4.64386   3.64458   2.25518   0.11562  -2.91533  -7.77921 -14.68094
##  [8] -20.35397 -30.41538 -41.57497 -52.89343 -67.04642
## attr(,"gradient")
##            b1       b2       b3
##  [1,]  9.9519  -8.9615  0.89615
##  [2,] 10.8846  -9.6998  1.93997
```

```
##  [3,] 11.8932 -10.4787  3.14361
##  [4,] 12.9816 -11.2964  4.51856
##  [5,] 14.1537 -12.1504  6.07520
##  [6,] 15.4128 -13.0373  7.82235
##  [7,] 16.7621 -13.9524  9.76668
##  [8,] 18.2040 -14.8902 11.91213
##  [9,] 19.7406 -15.8437 14.25933
## [10,] 21.3730 -16.8050 16.80496
## [11,] 23.1016 -17.7647 19.54122
## [12,] 24.9256 -18.7127 22.45528
```

```
myfn <- function(p, resfn=rjfn){
  val <- resss(p, resfn)
}

p <- c(b1 = 2, b2=2, b3=1)

a1 <- optim(p, myfn, control=list(trace=0))
a1
```

```
## $par
##     b1     b2     b3
## 1.9618 4.9092 3.1358
##
## $value
## [1] 2.5873
##
## $counts
## function gradient
##      200       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

We can also embed the function directly.

```
a2 <- optim(p, function(p,resfn=rjfn){resss(p,resfn)}, control=list(trace=0))
a2
```

```
## $par
##     b1     b2     b3
## 1.9618 4.9092 3.1358
##
## $value
## [1] 2.5873
##
## $counts
## function gradient
##      200       NA
##
## $convergence
## [1] 0
##
```

```
## $message
## NULL
```

Need more extensive discussion of Simplify??

Using derivative functions to generate gradient functions

?? from a different vignette, need to integrate

One of the common needs for optimization computations is the availability of accurate gradients of the objective functions. While differentiation is relatively straightforward, it is tedious and error-prone.

At 2015-1-24, I have not determined how to automate the use the output of the derivatives generated by **nlsr** to create a working gradient function. However, the following write-up shows how such a function can be generated in a semi-automatic way.

We use an example that appeared on the R-help mailing list on Jan 14, 2015. Responses by Ravi Varadhan and others, along with some modification I made, gave the following negative log likelihood function to be minimized.

```
require(nlsr)

y=c(5,11,21,31,46,75,98,122,145,165,195,224,245,293,321,330,350,420) # data set

Nweibull2 <- function(x,prm){
  la <- prm[1]
  al <- prm[2]
  be<- prm[3]
  val2 <- la*be*(x/al)^(be-1)* exp( (x/al)^be+la*al*(1-exp((x/al)^be) ) )
  val2
}
LL2J <- function(par,y) {
R = Nweibull2(y,par)
-sum(log(R))
}
```

We want the gradient of LL3J() with respect to *par*, and first compute the derivatives of Nweibull2() with respect to the paramters *prm*

We start with the central expression in Nweibull2() and compute its partial derivatives. The expression is:

```
  la*be*(x/al)^(be-1)* exp( (x/al)^be+la*al*(1-exp((x/al)^be) ) )
```

```
# Put in the main expression for the Nweibull pdf.
## we generate the three gradient components
g1n <- nlsDeriv(~ la*be*(x/al)^(be-1)* exp( (x/al)^be+la*al*(1-exp((x/al)^be) ) ), "la")
g1n
```

```
## la * be * (x/al)^(be - 1) * (exp((x/al)^be + la * al * (1 - exp((x/al)^be))) *
##     (al * (1 - exp((x/al)^be)))) + be * (x/al)^(be - 1) * exp((x/al)^be +
##     la * al * (1 - exp((x/al)^be)))
```

```
g2n <- nlsDeriv(~ la*be*(x/al)^(be-1)* exp( (x/al)^be+la*al*(1-exp((x/al)^be) ) ), "al")
g2n
```

```
## la * be * (x/al)^(be - 1) * (exp((x/al)^be + la * al * (1 - exp((x/al)^be))) *
##     (be * (x/al)^(be - 1) * -(x/al^2) + (la * al * -(exp((x/al)^be) *
##         (be * (x/al)^(be - 1) * -(x/al^2))) + la * (1 - exp((x/al)^be))))) +
##     la * be * ((be - 1) * (x/al)^(be - 1 - 1) * -(x/al^2)) *
```

```
##         exp((x/al)^be + la * al * (1 - exp((x/al)^be)))
```

```
g3n <- nlsDeriv(~ la*be*(x/al)^(be-1)* exp( (x/al)^be+la*al*(1-exp((x/al)^be) ) ), "be")
g3n
```

```
## la * be * (x/al)^(be - 1) * (exp((x/al)^be + la * al * (1 - exp((x/al)^be))) *
##     ((x/al)^be * log(x/al) + la * al * -(exp((x/al)^be) * ((x/al)^be *
##         log(x/al)))) + (la * be * ((x/al)^(be - 1) * log(x/al)) +
##     la * (x/al)^(be - 1)) * exp((x/al)^be + la * al * (1 - exp((x/al)^be)))
```

By copying and pasting the output above into a function structure, we get Nwei2g() below.

```
Nwei2g <- function(x, prm){
  la <- prm[1]
  al <- prm[2]
  be<- prm[3]
g1v <- la * be * (x/al)^(be - 1) * (exp((x/al)^be + la * al * (1 - exp((x/al)^be))) *
    (al * (1 - exp((x/al)^be)))) + be * (x/al)^(be - 1) * exp((x/al)^be +
    la * al * (1 - exp((x/al)^be)))

g2v <- la * be * (x/al)^(be - 1) * (exp((x/al)^be + la * al * (1 - exp((x/al)^be))) *
    (be * (x/al)^(be - 1) * -(x/al^2) + (la * al * -(exp((x/al)^be) *
        (be * (x/al)^(be - 1) * -(x/al^2))) + la * (1 - exp((x/al)^be))))) +
    la * be * ((be - 1) * (x/al)^(be - 1 - 1) * -(x/al^2)) *
        exp((x/al)^be + la * al * (1 - exp((x/al)^be)))

g3v <- la * be * (x/al)^(be - 1) * (exp((x/al)^be + la * al * (1 - exp((x/al)^be))) *
    ((x/al)^be * log(x/al) + la * al * -(exp((x/al)^be) * ((x/al)^be *
        log(x/al)))) + (la * be * ((x/al)^(be - 1) * log(x/al)) +
    la * (x/al)^(be - 1)) * exp((x/al)^be + la * al * (1 - exp((x/al)^be)))
gg <- matrix(data=c(g1v, g2v, g3v), ncol=3)
}
```

We can check this gradient functionusing the grad() function from package **numDeriv**.

```
start1 <- c(lambda=.01,alpha=340,beta=.8)
start2 <- c(lambda=.01,alpha=340,beta=.7)

require(numDeriv)
ganwei <- Nwei2g(y, prm=start1)

require(numDeriv)
Nw <- function(x, y) {
   Nweibull2(y, x)
} # to allow grad() to work

gnnwei <- matrix(NA, nrow=length(y), ncol=3)
for (i in 1:length(y)){
   gnrow <- grad(Nw, x=start1, y=y[i])
   gnnwei[i,] <- gnrow
}
gnnwei
```

```
##             [,1]        [,2]        [,3]
##  [1,]  1.5080091  7.5763e-06 -4.4573e-02
##  [2,]  1.0470119  4.3477e-06 -2.1663e-02
##  [3,]  0.6473921  1.6572e-06 -7.3707e-03
```

```
##  [4,]  0.4021585  1.7784e-07 -9.4940e-04
##  [5,]  0.1635446 -1.0061e-06  3.5893e-03
##  [6,] -0.0815624 -1.6713e-06  6.0571e-03
##  [7,] -0.1689654 -1.5386e-06  5.8709e-03
##  [8,] -0.2048835 -1.1819e-06  5.0149e-03
##  [9,] -0.2077443 -7.9701e-07  4.0222e-03
## [10,] -0.1955391 -4.9171e-07  3.1859e-03
## [11,] -0.1644138 -1.3523e-07  2.1110e-03
## [12,] -0.1297028  8.2595e-08  1.3249e-03
## [13,] -0.1055059  1.7196e-07  9.0488e-04
## [14,] -0.0598567  2.2613e-07  3.1939e-04
## [15,] -0.0406518  2.0242e-07  1.4872e-04
## [16,] -0.0355949  1.9079e-07  1.1187e-04
## [17,] -0.0261120  1.6182e-07  5.3031e-05
## [18,] -0.0075317  6.8245e-08 -1.1995e-05
```

```
ganwei
```

```
##             [,1]        [,2]        [,3]
##  [1,]  1.5080091  7.5763e-06 -4.4573e-02
##  [2,]  1.0470119  4.3477e-06 -2.1663e-02
##  [3,]  0.6473921  1.6572e-06 -7.3707e-03
##  [4,]  0.4021585  1.7784e-07 -9.4940e-04
##  [5,]  0.1635446 -1.0061e-06  3.5893e-03
##  [6,] -0.0815624 -1.6713e-06  6.0571e-03
##  [7,] -0.1689654 -1.5386e-06  5.8709e-03
##  [8,] -0.2048835 -1.1819e-06  5.0149e-03
##  [9,] -0.2077443 -7.9701e-07  4.0222e-03
## [10,] -0.1955391 -4.9171e-07  3.1859e-03
## [11,] -0.1644138 -1.3523e-07  2.1110e-03
## [12,] -0.1297028  8.2595e-08  1.3249e-03
## [13,] -0.1055059  1.7196e-07  9.0488e-04
## [14,] -0.0598567  2.2613e-07  3.1939e-04
## [15,] -0.0406518  2.0242e-07  1.4872e-04
## [16,] -0.0355949  1.9079e-07  1.1187e-04
## [17,] -0.0261120  1.6182e-07  5.3031e-05
## [18,] -0.0075317  6.8245e-08 -1.1995e-05
```

```
cat("max(abs(gnnwei - ganwei))= ",   max(abs(gnnwei - ganwei)),"\n")
```

```
## max(abs(gnnwei - ganwei))=  2.8041e-11
```

Now we can build the gradient of the objective function. This requires an application of the chain rule to the summation of logs of the elements of the quantity R. Since the derivative of log(R) w.r.t. R is simply 1/R, this is relatively simple. However, I have not found how to automate this.

```
## and now we can build the gradient of LL2J
LL2Jg <- function(prm, y) {
    R = Nweibull2(y,prm)
    gNN <- Nwei2g(y, prm)
#    print(str(gNN)
    gL <- - as.vector( t(1/R) %*% gNN)
}
# test
gaLL2J <- LL2Jg(start1, y)
gaLL2J
```

```
## [1] 3365.78244   -0.01441  -18.63351
```

```
gnLL2J <- grad(LL2J, start1, y=y)
gnLL2J
```

```
## [1] 3365.78244   -0.01441  -18.63351
```

```
cat("max(abs(gaLL2J-gnLL2J))= ", max(abs(gaLL2J-gnLL2J)), "\n" )
```

```
## max(abs(gaLL2J-gnLL2J))=  1.8254e-08
```

Appendix A: Providing exogenous data

These examples show dotargs do NOT work for any of nlsr, nls, or minpack.lm. Use of a dataframe or local (calling) environment objects does work in all.

```
## Data variables:[1] "y"  "tt"
## Are the variables present in the current working environment?
## y  : present= TRUE
## tt  : present= TRUE
```

```
## Data variables:[1] "y"  "tt"
## Are the variables present in the current working environment?
## y  : present= TRUE
## tt  : present= TRUE
```

```
## dots:$y
##  [1]  5.308  7.240  9.638 12.866 17.069 23.192 31.443 38.558 50.156 62.948
## [11] 75.995 91.972
##
## $tt
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
##
## names in dots:[1] "y"  "tt"
## Data variables:[1] "y"  "tt"
## Are the variables present in the dot args?
## y  : present= TRUE
## tt  : present= TRUE
```

```
## Nonlinear regression model
##   model: y ~ 100 * b1/(1 + 10 * b2 * exp(-0.1 * b3 * tt))
##    data: parent.frame()
##   b1   b2   b3
## 1.96 4.91 3.14
##  residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 4
## Achieved convergence tolerance: 2.17e-07
```

```
## Nonlinear regression model
##   model: y ~ 100 * b1/(1 + 10 * b2 * exp(-0.1 * b3 * tt))
##    data: mydata
##   b1   b2   b3
## 1.96 4.91 3.14
##  residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 4
## Achieved convergence tolerance: 2.17e-07
```

```
## Nonlinear regression model
##   model: y ~ 100 * b1/(1 + 10 * b2 * exp(-0.1 * b3 * tt))
##    data: mydata
##   b1   b2   b3
## 1.96 4.91 3.14
##  residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 4
## Achieved convergence tolerance: 2.17e-07
```

```
## nlsr object: x
## residual sumsquares =  2.5873  on  12 observations
##     after  6    Jacobian and  7 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08  -6.418e-12       130.1
## b2               4.90916        0.1688      29.08  3.284e-10  -1.979e-12       6.165
## b3                3.1357       0.06863      45.69  5.768e-12   1.243e-11       2.735
```

```
## Error in nlxb(formula = hobsc, start = ste, y = ydata, tt = ttdata) :
##   unused arguments (y = ydata, tt = ttdata)
```

```
## Try error
```

```
## Try error
```

```
## Nonlinear regression model
##   model: y ~ 100 * b1/(1 + 10 * b2 * exp(-0.1 * b3 * tt))
##    data: parent.frame()
##   b1   b2   b3
## 1.96 4.91 3.14
##  residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 4
## Achieved convergence tolerance: 1.49e-08
```

```
## Nonlinear regression model
##   model: y ~ 100 * b1/(1 + 10 * b2 * exp(-0.1 * b3 * tt))
##    data: mydata
##   b1   b2   b3
## 1.96 4.91 3.14
##  residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 4
## Achieved convergence tolerance: 1.49e-08
```

```
## Nonlinear regression model
##   model: y ~ 100 * b1/(1 + 10 * b2 * exp(-0.1 * b3 * tt))
##    data: mydata
##   b1   b2   b3
## 1.96 4.91 3.14
##  residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 4
## Achieved convergence tolerance: 1.49e-08
```

References

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

Hartley, H. O. 1961. "The Modified Gauss-Newton Method for Fitting of Nonlinear Regression Functions by Least Squares." *Technometrics* 3: 269–80.

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

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

Nash, John C. 1977. "Minimizing a Nonlinear Sum of Squares Function on a Small Computer." *Journal of the Institute for Mathematics and Its Applications* 19: 231–37.