Package ‘nlsr’

August 20, 2021

Type Package
Title Functions for Nonlinear Least Squares Solutions
Version 2021.8.19
Date 2021-08-19
Maintainer John C Nash <nashjc@uottawa.ca>
Description Provides tools for working with nonlinear least squares problems. It is intended to eventually supersede the ‘nls()’ function in the R distribution. For example, 'nls()' specifically does NOT deal with small or zero residual problems as its Gauss-Newton method frequently stops with 'singular gradient' messages. 'nlsr' is based on the now-deprecated package 'nlmrt', and has refactored functions and R-language symbolic derivative features.
License GPL-2
Depends R (>= 3.0)
Imports digest
Suggests minpack.lm, optimx, numDeriv, knitr, rmarkdown, markdown, Ryacas, Deriv
VignetteBuilder knitr
NeedsCompilation no
Author John C Nash [aut, cre], Duncan Murdoch [aut]
Repository CRAN
Date/Publication 2021-08-20 08:40:02 UTC

R topics documented:

  nlsr-package .................................................. 2
  coef.nlsr .................................................... 6
  dex ........................................................... 7
  findSubexps .................................................. 8
  model2rjfun .................................................. 9
Tools for solving nonlinear least squares problems. UNDER DEVELOPMENT.

Description

The package provides some tools related to using the Nash variant of Marquardt’s algorithm for nonlinear least squares.

Details

<table>
<thead>
<tr>
<th>Package:</th>
<th>nlsr</th>
<th>Type:</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version:</td>
<td>1.0</td>
<td>Date:</td>
<td>2012-03-05</td>
</tr>
<tr>
<td>License:</td>
<td>GPL-2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This package includes methods for solving nonlinear least squares problems specified by a modeling expression and given a starting vector of named parameters. Note: You must provide an expression of the form lhs ~ rhsexpression so that the residual expression rhsexpression - lhs can be computed. The expression can be enclosed in quotes, and this seems to give fewer difficulties with R. Data variables must already be defined, either within the parent environment or else in the dot-arguments. Other symbolic elements in the modeling expression must be standard functions or else parameters that are named in the start vector.

The main functions in nlsr are:

- **nlfb**: Nash variant of the Marquardt procedure for nonlinear least squares, with bounds constraints, using a residual and optionally Jacobian described as R functions.

- **nlxb**: Nash variant of the Marquardt procedure for nonlinear least squares, with bounds constraints, using an expression to describe the residual via an R modeling expression. The Jacobian is computed via symbolic differentiation.
wrapnlsr Uses nlx to solve nonlinear least squares then calls nls() to create an object of type nls.

model2rjfun returns a function with header function(prm), which evaluates the residuals (and if jacobian is TRUE the Jacobian matrix) of the model at prm. The residuals are defined to be the right hand side of modelformula minus the left hand side.

model2sgrfun returns a function with header function(prm), which evaluates the sum of squared residuals (and if gradient is TRUE the gradient vector) of the model at prm.

modelexpr returns the expression used to calculate the vector of residuals (and possibly the Jacobian) used in the previous functions.

Author(s)

John C Nash and Duncan Murdoch
Maintainer: <nashjc@uottawa.ca>

References


See Also

nls

Examples

```r
rm(list=ls())
# require(nlsr)

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 nlx, 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")
weeddata1 <- data.frame(y=ydat, tt=tdata)
weeddata2 <- data.frame(y=1.5*ydat, tt=tdata)
anlxb1 <- try(nlx(eunsc, start=start1, trace=traceval, data=weeddata1))
print(anlxb1)
```
# illustrate predict
newdta <- colMeans(weeddata1)
newdta["tt"]<=-25 # This only works for 1D example -- CAUTION
predn <- predict(anlxb1, as.list(newdta))
print(predn)

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

escal <- y ~ 100*b1/(1+10*b2*exp(-0.1*b3*tt))
suneasy <- c(b1=200, b2=50, b3=0.3)
ssceasy <- c(b1=2, b2=5, b3=3)
st1scal <- c(b1=100, b2=10, b3=0.1)

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 # needed for nlfb to function seamlessly with nlxb
  return(jj)
}

cat("try nlfb\n")
st <- c(b1=1, b2=1, b3=1)
low <- -Inf
up <- Inf

## Not run:
ans1 <- nlfb(st, shobbs.res, shobbs.jac, trace=traceval)
ans1
cat("No jacobian function -- use internal approximation\n")
ansln <- nlfb(st, shobbs.res, trace=TRUE, control=list(watch=TRUE)) # NO jacfn
ansln
# tmp <- readline("Try with bounds at 2")
time2 <- system.time(ans2 <- nlfb(st, shobbs.res, shobbs.jac, upper=c(2,2,2),
                        trace=traceval))
ans2
time2

## End(Not run) # end dontrun

cat("BOUNDS")
st2s <- c(b1=1, b2=1, b3=1)

## Not run:
an1qb1 <- try(nlxb(escal, start=st2s, trace=traceval, data=weeddata1,
                  lower=c(0,0,0), upper=c(2, 6, 3), control=list(watch=FALSE)))
print(an1qb1)
tmp <- readline("next")
ans2 <- nlfb(st2s,shobbs.res, shobbs.jac, lower=c(0,0,0), upper=c(2, 6, 3),
               trace=traceval, control=list(watch=FALSE))
print(ans2)

cat("BUT... nls() seems to do better from the TRACE information\n")
anlsb <- nls(escal, start=st2s, trace=traceval, data=weeddata1, lower=c(0,0,0),
            upper=c(2,6,3), algorithm='port')
cat("However, let us check the answer\n")
print(anlsb)
cat("BUT...crossprod(resid(anlsb))="
crossprod(resid(anlsb))"\n")

## End(Not run) # end dontrun

tmp <- readline("next")
cat("Try wrapnlsr\n")
traceval <- TRUE
# 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)
escal <- y ~ 100*b1/(1+10*b2*exp(-0.1*b3*tt))
up1 <- c(2,6,3)
up2 <- c(1, 5, 9)

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

## Not run:
an1w <- try(wrapnlsr(escal, start=start1, trace=traceval, data=weeddata1))
print(an1w)

cat("BOUNDED wrapnlsr\n")

an1wb <- try(wrapnlsr(escal, start=start1, trace=traceval, data=weeddata1, upper=up1))
print(an1wb)

cat("BOUNDED wrapnlsr\n")

an2wb <- try(wrapnlsr(escal, start=start1, trace=traceval, data=weeddata1, upper=up2))
print(an2wb)

cat("TRY MASKS ONLY\n")

an1xm3 <- try(nlxb(escal, start1, trace=traceval, data=weeddata1,
masked=c("b3")))
printsum(an1xm3)

an1xm1 <- try(nlxb(escal, start1, trace=traceval, data=weeddata1,
masked=c("b1")))
printsum(an1xm1)

# Need to check when all parameters masked.

---

**coef.nlsr**  
*Output model coefficients for nlsr object.*

**Description**

`coef.nlsr` extracts and displays the coefficients for a model estimated by `nlxb` or `nlfb` in the `nlsr` structured object.
Usage

```r
## S3 method for class 'nlsr'
coef(object, ...)
```

Arguments

- `object`: An object of class 'nlsr'
- `...`: Any data needed for the function. We do not know of any!

Details

coeef.nlsr extracts and displays the coefficients for a model estimated by nlxb or nlfb.

Value

returns the coefficients from the nlsr object.

Author(s)

John C Nash <nashjc@uottawa.ca>

See Also

Function `nls()`, packages `optim` and `optimx`.

dex  

`Calculate expression for derivative calculations.`

Description

Converts input to an expression suitable for use in `nlsDeriv` and related functions.

Usage

```r
dex(x, do_substitute = NA, verbose = FALSE)
```

Arguments

- `x`: An expression represented in a variety of ways. See Details.
- `do_substitute`: Whether to use the expression passed as `x`, or to evaluate it and use its value.
- `verbose`: Print messages describing the process.
Details

If do_substitute is NA, the following rules are used:

1. An attempt is made to evaluate x. If that fails, the expression is used.
2. If the evaluation succeeds and the value is a character vector, it is parsed.
3. If the value is not a character vector and the expression is a single name, the value is used.
4. Otherwise, the expression is used.

Once the expression is determined it may be simplified, by extracting the language object from a length-one expression vector, or the right-hand-side from a formula.

Normally a warning will be issued if x is a formula containing a left-hand side. To suppress this, wrap the formula in expression(), or pass it as a character string to be parsed.

Value

An expression or language object suitable as input to nlsDeriv and related functions.

Author(s)

Duncan Murdoch

Examples

```r
aa <- dex(~ x^2)
aa
str(aa)
bb <- dex(expression(x^2))
bb
str(bb)
cc <- dex("x^2")
cc
str(cc)
```

Description

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

Usage

```r
findSubexprs(expr, simplify = FALSE, tag = ".expr", verbose = FALSE, ...)
```
Arguments

expr An expression vector or language object.
simplify Whether to call \texttt{nlsSimplify} on each subexpression before looking for common subexpressions.
tag The prefix to use for locally created variables.
verbose If \texttt{TRUE}, diagnostics are printed as simplifications are recognized.
... Additional parameters to pass to \texttt{nlsSimplify}. Used only if \texttt{simplify} is \texttt{TRUE}.

Details

This function identifies all repeated subexpressions in an expression vector, and stores them in locally created variables. It is used by \texttt{fnDeriv} to share common subexpressions between expression evaluations and gradient evaluations, for example.

If \texttt{simplify} is \texttt{TRUE}, the assumptions behind the simplifications done by \texttt{nlsSimplify} must be valid for the result to match the input. With the default simplifications, this means that all variables should take finite real values.

Value

A language object which evaluates to an expression vector which would evaluate to the same result as the original vector with less duplicated code but more storage of intermediate results.

Author(s)

Duncan Murdoch

See Also

\texttt{deriv} in the \texttt{stats} package, \texttt{nlsSimplify}

Examples

\begin{verbatim}
findSubexprs(expression(x^2, x-y, y^2-x^2))
\end{verbatim}

\begin{verbatim}
model2rjfun
\end{verbatim}

\textit{Create functions to calculate the residual vector or the sum of squares, possibly with derivatives.}

Description

These functions create functions to evaluate residuals or sums of squares at particular parameter locations.
Usage

model2rjfun(modelformula, pvec, data = NULL, jacobian = TRUE, testresult = TRUE, ...)  
model2ssgrfun(modelformula, pvec, data = NULL, gradient = TRUE, testresult = TRUE, ...)  
modelexpr(fun)

Arguments

modelformula  A formula describing a nonlinear regression model.
pvec  A vector of parameters.
data  A dataframe, list or environment holding data used in the calculation.
jacobian  Whether to compute the Jacobian matrix.
gradient  Whether to compute the gradient vector.
testresult  Whether to test the function by evaluating it at pvec.
fun  A function produced by one of model2rjfun or model2ssgrfun.
...  Dot arguments, that is, arguments that may be supplied by name = value to supply information needed to compute specific quantities in the model.

Details

If pvec does not have names, the parameters will have names generated in the form ‘p_<n>’, e.g. p_1, p_2. Names that appear in pvec will be taken to be parameters of the model.

The data argument may be a dataframe, list or environment, or NULL. If it is not an environment, one will be constructed using the components of data with parent environment set to be the environment of modelformula.

Value

model2rjfun returns a function with header function(prm), which evaluates the residuals (and if jacobian is TRUE the Jacobian matrix) of the model at prm. The residuals are defined to be the right hand side of modelformula minus the left hand side.

model2ssgrfun returns a function with header function(prm), which evaluates the sum of squared residuals (and if gradient is TRUE the gradient vector) of the model at prm.

modelexpr returns the expression used to calculate the vector of residuals (and possibly the Jacobian) used in the previous functions.

Author(s)

John Nash and Duncan Murdoch

See Also

nls
Examples

```r
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 ~ b1/(1 + b2 * exp(-1 * b3 * tt))
p <- c(b1 = 1, b2 = 1, b3 = 1)
rjfn <- model2rjfun(f, p, data = mydata)
rjfn(p)
myexp <- modelexpr(rjfn)
cat("myexp:")
print(myexp)

ssgrfn <- model2ssgrfun(f, p, data = mydata)
ssgrfn(p)
```

Description

Given a nonlinear model expressed as a vector valued residual function `resfn` and a start vector of parameter values for that function, 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. This is a restructured version of a function by the same name from package `nlmrt` which is now deprecated.

Usage

```r
nlfb(start, resfn, jacfn=NULL, trace=FALSE, lower=-Inf, upper=Inf,
     maskidx=NULL, weights=NULL, data=NULL, control, ...)
```

Arguments

- `resfn`: A function that evaluates the residual vector for computing the elements of the sum of squares function at the set of parameters `start`. Where this function is created by actions on a formula or expression in `nlxb`, this residual vector will be created by evaluation of the ’model - data’, rather than the conventional ’data - model’ approach. The sum of squares is the same.

- `jacfn`: A function that evaluates the Jacobian of the sum of squares function, that is, the matrix of partial derivatives of the residuals with respect to each of the parameters. If NULL (default), uses an approximation. The Jacobian MUST be returned as the attribute “gradient” of this function, allowing `jacfn` to have the same name and be the same code block as `resfn`, which may permit some efficiencies of computation.
A named parameter vector. For our example, we could use `start=c(b1=1, b2=2.345, b3=0.123)`.

`nls()` takes a list, and that is permitted here also.

Logical TRUE if we want intermediate progress to be reported. Default is FALSE.

Lower bounds on the parameters. If a single number, this will be applied to all parameters. Default -Inf.

Upper bounds on the parameters. If a single number, this will be applied to all parameters. Default Inf.

Vector if indices of the parameters to be masked. These parameters will NOT be altered by the algorithm. Note that the mechanism here is different from that in `nls` which uses the names of the parameters.

A vector of fixed weights. The objective function that will be minimized is the sum of squares where each residual is multiplied by the square root of the corresponding weight. Default (NULL) implies unit weights.

Data frame of variables used by `resfn` and `jacf` to compute the required residuals and Jacobian.

A list of controls for the algorithm. These are:

- `watch` Monitor progress if TRUE. Default is FALSE.
- `phi` Default is phi=1, which adds phi*Identity to Jacobian inner product.
- `lambda` Initial Marquardt adjustment (Default 0.0001). Odd spelling is deliberate.
- `offset` Shift to test for floating-point equality. Default is 100.
- `laminc` Factor to use to increase lambda. Default is 10.
- `lamdec` Factor to use to decrease lambda/laminc. Default lamdec=4.
- `femax` Maximum function (sum of squares) evaluations. Default is 10000, which is extremely aggressive.
- `jemax` Maximum number of Jacobian evaluations. Default is 5000.
- `ndstep` Stepsize to use to computer numerical Jacobian approximation. Default is 1e-7.
- `rofftest` Default is TRUE. Use a termination test of the relative offset orthogonality type. Useful for nonlinear regression problems.
- `smallssstest` Default is TRUE. Exit the function if the sum of squares falls below (100 * .Machine$double.eps)^4 times the initial sumsquares. This is a test for a “small” sum of squares, but there are problems which are very extreme for which this control needs to be set FALSE.

Any data needed for computation of the residual vector from the expression `rhsexpression - lhsvar`. Note that this is the negative of the usual residual, but the sum of squares is the same. It is not clear how the dot variables should be used, since data should be in 'data'.

Details

`nlfb` attempts to solve the nonlinear sum of squares problem by using a variant of Marquardt’s approach to stabilizing the Gauss-Newton method using the Levenberg-Marquardt adjustment. This is explained in Nash (1979 or 1990) in the sections that discuss Algorithm 23.
In this code, we solve the (adjusted) Marquardt equations by use of the \texttt{qr.solve()} function. Rather than forming the $J'J + \lambda D$ matrix, we augment the $J$ matrix with extra rows and the $y$ vector with null elements.

\textbf{Value}

A list of the following items:

- \texttt{coefficients} A named vector giving the parameter values at the supposed solution.
- \texttt{ssquares} The sum of squared residuals at this set of parameters.
- \texttt{resid} The residual vector at the returned parameters.
- \texttt{jacobian} The jacobian matrix (partial derivatives of residuals w.r.t. the parameters) at the returned parameters.
- \texttt{feval} The number of residual evaluations (sum of squares computations) used.
- \texttt{jeval} The number of Jacobian evaluations used.

\textbf{Author(s)}

John C Nash <nashjc@uottawa.ca>

\textbf{References}


See Also

Function \texttt{nls()}, packages \texttt{optim} and \texttt{optimx}.

\textbf{Examples}

\begin{verbatim}
cat("See examples in nls-package.Rd\n")
\end{verbatim}
Usage

nlsDeriv(expr, name, derivEnv = sysDerivs, do_substitute = FALSE, verbose = FALSE, ...)
codeDeriv(expr, namevec, hessian = FALSE, derivEnv = sysDerivs, 
do_substitute = FALSE, verbose = FALSE, ...)
fnDeriv(expr, namevec, args = all.vars(expr), env = environment(expr),
   do_substitute = FALSE, verbose = FALSE, ...)
newDeriv(expr, deriv, derivEnv = sysDerivs)
sysDerivs

Arguments

expr An expression represented in a variety of ways. See Details.
name The name of the variable with respect to which the derivative will be computed.
derivEnv The environment in which derivatives are stored.
do_substitute If TRUE, use substitute to get the expression passed as expr, otherwise evaluate it.
verbose If TRUE, then diagnostic output will be printed as derivatives and simplifications are recognized.
... Additional parameters which will be passed to codeDeriv from fnDeriv, and to nlsSimplify from nlsDeriv and codeDeriv.
namelnec Character vector giving the variable names with respect to which the derivatives will be taken.
hessian Logical indicator of whether the 2nd derivatives should also be computed.
deriv An expression giving the derivative of the function call in expr.
args Desired arguments for the function. See Details below.
env The environment to be attached to the created function. If NULL, the caller’s frame is used.

Details

Functions nlsDeriv and codeDeriv are designed as replacements for the stats package functions D and deriv respectively, though the argument lists do not match exactly.

The nlsDeriv function computes a symbolic derivative of an expression or language object. Known derivatives are stored in derivEnv; the default sysDerivs contains expressions for all of the derivatives recognized by deriv, but in addition allows differentiation with respect to any parameter where it makes sense. It also allows the derivative of abs and sign, using an arbitrary choice of 0 at the discontinuities.

The codeDeriv function computes an expression for efficient calculation of the expression value together with its gradient and optionally the Hessian matrix.

The fnDeriv function wraps the codeDeriv result in a function. If the args are given as a character vector (the default), the arguments will have those names, with no default values. Alternatively, a custom argument list with default values can be created using alist; see the example below.
The `expr` argument will be converted to a language object using `dex` (but note the different default for `do_substitute`). Normally it should be a formula with no left hand side, e.g. `~ x^2`, or an expression vector e.g. `expression(x, x^2, x^3)`, or a language object e.g. `quote(x^2)`. In `codeDeriv` and `fnDeriv` the expression vector must be of length 1.

The `newDeriv` function is used to define a new derivative. The `expr` argument should match the header of the function as a call to it (e.g. as in the help pages), and the `deriv` argument should be an expression giving the derivative, including calls to `D(arg)`, which will not be evaluated, but will be substituted with partial derivatives of that argument with respect to `name`. See the examples below.

If `expr` or `deriv` is missing in a call to `newDeriv()`, it will return the currently saved derivative record from `derivEnv`. If `name` is missing in a call to `nlsDeriv` with a function call, it will print a message describing the derivative formula and return `NULL`.

To handle functions which act differently if a parameter is missing, code the default value of that parameter to `.MissingVal`, and give a derivative that is conditional on `missing()` applied to that parameter. See the derivatives of `"-"` and `"+"` in the file `derivs.R` for an example.

**Value**

If `expr` is an expression vector, `nlsDeriv` and `nlsSimplify` return expression vectors containing the response. For formulas or language objects, a language object is returned.

codeDeriv always returns a language object.

`fnDeriv` returns a closure (i.e. a function).

`nlsDeriv` returns the symbolic derivative of the expression.

`newDeriv` with `expr` and `deriv` specified is called for the side effect of recording the derivative in `derivEnv`. If `expr` is missing, it will return the list of names of functions for which derivatives are recorded. If `deriv` is missing, it will return its record for the specified function.

**Note**

`newDeriv(expr, deriv, ...)` will issue a warning if a different definition for the derivative exists in the derivative table.

**Author(s)**

Duncan Murdoch

**See Also**

`deriv, nlsSimplify`

**Examples**

```r
newDeriv()
newDeriv(sin(x))
nlsDeriv(~ sin(x+y), "x")

f <- function(x) x^2
newDeriv(f(x), 2*x*D(x))
nlsDeriv(~ f(abs(x)), "x")
```
nlsDeriv(~ pnorm(x, sd=2, log = TRUE), "x")
fnDeriv(~ pnorm(x, sd = sd, log = TRUE), "x")
f <- fnDeriv(~ pnorm(x, sd = sd, log = TRUE), "x", args = alist(x =, sd = 2))
f
f(1)
100*(f(1.01) - f(1)) # Should be close to the gradient
# The attached gradient attribute (from f(1.01)) is
# meaningless after the subtraction.

# Multiple point example
xvals <- c(1, 3, 4.123)
print(f(xvals))

# Getting a hessian matrix
f2 <- ~ (x-2)^3*y - y^2
mydf2 <- fnDeriv(f2, c("x","y"), hessian=TRUE)
# display the resulting function
print(mydf2)
x <- c(1, 2)
y <- c(0.5, 0.1)
evalmydf2 <- mydf2(x, y)
print(evalmydf2)
# the first index of the hessian attribute is the point at which we want the hessian
hmat1 <- as.matrix(attr(evalmydf2,"hessian")[1,,])
print(hmat1)
hmat2 <- as.matrix(attr(evalmydf2,"hessian")[2,,])
print(hmat2)

nlsSimplify

Description

nlsSimplify simplifies expressions according to rules specified by newSimplification.

Usage

nlsSimplify(expr, simpEnv = sysSimplifications, verbose = FALSE)
newSimplification(expr, test, simplification, do_eval = FALSE,
simpEnv = sysSimplifications)
sysSimplifications
isFALSE(x)
isZERO(x)
isONE(x)
isMINUSONE(x)
isCALL(x, name)
**nlsSimplify**

**Arguments**

- **expr** An expression to simplify; this should be a language object.
- **simpEnv** An environment holding the registered simplifications.
- **verbose** If TRUE, diagnostics are printed as simplifications are recognized.
- **test** An expression giving a test to apply to decide whether this simplification applies.
- **simplification** The new expression to use to replace the original one.
- **do_eval** Whether to evaluate the new expression (to give an even newer expression) to use as the simplification.
- **x** An expression to test.
- **name** The name of a function as a character string.

**Details**

The `nlsSimplify` function uses simple rules to simplify expressions. The simplification is aimed at the needs of this package, so the built-in simplifications assume that variables and expressions have finite real values. For example, `0*expr` will simplify to `0` regardless of the value of `expr`. (The name is `nlsSimplify` to avoid a clash with the `Simplify` function in the `Deriv` package.)

`newSimplification` adds a new simplification pattern to the registered collection. The tests are applied to function calls with the same function and number of parameters, in order as specified. Users may specify their own environment (perhaps parented by `sysSimplifications`) to hold rules if they wish to override the standard rules.

The `isFALSE`, `isZERO`, `isONE`, and `isMINUSONE` functions are simple functions to test whether expressions are simple constants, similar to `isTRUE`.

The `isCALL` function tests whether an expression is a call to a particular function.

To handle functions which act differently depending on which arguments are present, `nlsSimplify` will simplify the expression `missing(.MissingVal)` to `TRUE`. This is used in the definition of the derivative for `x-y`, where the unary minus is seen as a missing `y` value.

**Value**

`nlsSimplify` returns a simplification of the expression, assuming that variables and functions take real values.

The `newSimplification` function is called for the side effect of recording a new simplification rule.

If `expr` or `deriv` is missing, `newSimplification()` will report on the currently saved simplifications in `simpEnv`.

**Note**

The `isFALSE` function was added to base R in version 3.5.0; starting with that version, `nlsr::isFALSE` is simply a copy of it.

**Author(s)**

Duncan Murdoch
See Also

`nlsDeriv`, which makes use of `nlsSimplify`.

Examples

```r
nlsSimplify(quote(a + 0))
nlsSimplify(quote(exp(1)), verbose = TRUE)
nlsSimplify(quote(sqrt(a + b)))  # standard rule
myrules <- new.env(parent = sysSimplifications)
newSimplification(sqrt(a), TRUE, a^0.5, simpEnv = myrules)
nlsSimplify(quote(sqrt(a + b)), simpEnv = myrules)
```

---

**nlxb**

*Nash variant of Marquardt nonlinear least squares solution via qr linear solver.*

Description

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. This is a restructured version of a function by the same name from package `nlmrt` which is now deprecated.

Usage

```r
nlxb(formula, start, trace=FALSE, data, lower=-Inf, upper=Inf,
masked=NULL, weights=NULL, control)
```

Arguments

- **formula**: This is a modeling formula of the form (as in `nls`) `lhsv~formula_for_rhs` for example, `y ~ b1/(1+b2*exp(-b3*tt))` You may also give this as a string. Note that the residuals are computed within this code using `residual <- rhsexpression ~ lhsvar` which is the negative of the usual choice, but the sum of squares is the same.
- **start**: A named parameter vector. For our example, we could use `start=c(b1=1,b2=2.345,b3=0.123)`
- **trace**: Logical `TRUE` if we want intermediate progress to be reported. Default is `FALSE`.
- **data**: A data frame containing the data of the variables in the formula. This data may, however, be supplied directly in the parent frame.
- **lower**: Lower bounds on the parameters. If a single number, this will be applied to all parameters. Default `-Inf`.
- **upper**: Upper bounds on the parameters. If a single number, this will be applied to all parameters. Default `Inf`. 
Character vector of quoted parameter names. These parameters will NOT be altered by the algorithm. Masks may also be defined by setting lower and upper bounds equal for the parameters to be fixed. Note that the starting parameter value must also be the same as the lower and upper bound value.

weights
A vector of fixed weights. The objective function that will be minimized is the sum of squares where each residual is multiplied by the square root of the corresponding weight. Default NULL implies unit weights.

control
A list of controls for the algorithm. These are:

- watch Monitor progress if TRUE. Default is FALSE.
- phi Default is phi=1, which adds phi*Identity to Jacobian inner product.
- lambda Initial Marquardt adjustment (Default 0.0001). Odd spelling is deliberate.
- offset Shift to test for floating-point equality. Default is 100.
- laminc Factor to use to increase lambda. Default is 10.
- lamdec Factor to use to decrease lambda is lamdec/laminc. Default lamdec=4.
- femax Maximum function (sum of squares) evaluations. Default is 10000, which is extremely aggressive.
- jemax Maximum number of Jacobian evaluations. Default is 5000.
- rofftest Default is TRUE. Use a termination test of the relative offset orthogonality type. Useful for nonlinear regression problems.
- smallsstest Default is TRUE. Exit the function if the sum of squares falls below \((100 \times .Machine$double.eps)^4\) times the initial sumsquares. This is a test for a “small” sum of squares, but there are problems which are very extreme for which this control needs to be set FALSE.

Details

nlxb attempts to solve the nonlinear sum of squares problem by using a variant of Marquardt’s approach to stabilizing the Gauss-Newton method using the Levenberg-Marquardt adjustment. This is explained in Nash (1979 or 1990) in the sections that discuss Algorithm 23.

In this code, we solve the (adjusted) Marquardt equations by use of the qr.solve(). Rather than forming the \(JJ + lambda*D\) matrix, we augment the \(J\) matrix with extra rows and the \(y\) vector with null elements.

Value

A list of the following items

- coefficients A named vector giving the parameter values at the supposed solution.
- ssquares The sum of squared residuals at this set of parameters.
- resid The residual vector at the returned parameters.
- jacobian The jacobian matrix (partial derivatives of residuals w.r.t. the parameters) at the returned parameters.
- feval The number of residual evaluations (sum of squares computations) used.
- jeval The number of Jacobian evaluations used.
Author(s)

John C Nash <nashjc@uottawa.ca>

References


See Also

Function nls(), packages optim and optimx.

Examples

```r
cat("See examples in nlsr-package.Rd\n")
```

predict.nlsr  

Predictions for models specified as a formula of style y ~ (something)

Description

Function to allow predictions from nonlinear models estimated with n1xb from package nlsr if the model is specified by a formula with the structure y ~ (something).

Usage

```r
## S3 method for class 'nlsr'
## S3 method for class 'nlsr'
predict(object, newdata,...)
```

Arguments

- **object**: The output object of executing n1xb to estimate the model. An object of class 'nlsr'.
- **newdata**: A named list containing the new data. This should be derived from a data frame of the same structure as the data used to estimate the model.
- **...**: Any data needed for the function. We do not know of any! This is NOT currently used.

Details

To be added.
Print method for an object of class `nlsr`.

### Description

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

### Usage

```r
## S3 method for class 'nlsr'
print(x, ...)
```

### Arguments

- **x**: An object of class `nlsr`
- **...**: Any data needed for the function. We do not know of any!

### Details

`printsum.nlsr` performs a print method for an object of class `nlsr` that has been created by a routine such as `nlfb` or `nlxb` for nonlinear least squares problems.

### Value

Invisibly returns the input object.

### Author(s)

John C Nash <nashjc@uottawa.ca>
**See Also**

Function `nls()`, packages `optim` and `optimx`.

---

**res**

*Evaluate residuals from an object of class nlsr.*

**Description**

Functions `nlfb` and `nlxb` return nonlinear least squares solution objects that include (weighted) residuals. If weights are present, the returned quantities are the square roots of the weights times the raw residuals.

**Usage**

`res(object)`

**Arguments**

- `object` An R object of class `nlsr`.

**Details**

`resgr` calls `resfn` to compute residuals and `jacfn` to compute the Jacobian at the parameters `prm` using external data in the dot arguments. It then computes the gradient using `t(Jacobian) . residuals`.

Note that it appears awkward to use this function in calls to optimization routines. The author would like to learn why.

**Value**

The numeric vector with the gradient of the sum of squares at the parameters.

**Author(s)**

John C Nash <nashjc@uottawa.ca>

**References**


**See Also**

Function `nls()`, packages `optim` and `optimx`. 
Examples

```r
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
}

st <- c(b1=1, b2=1, b3=1)
RG <- resgr(st, shobbs.res, shobbs.jac)
RG
```

---

**resgr**

*Compute gradient from residuals and Jacobian.*

**Description**

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`.

**Usage**

```r
resgr(prm, resfn, jacfn, ...)
```

**Arguments**

- **prm**: A parameter vector. For our example, we could use `start=c(b1=1, b2=2.345, b3=0.123)` However, the names are NOT used, only positions in the vector.
- **resfn**: A function to compute the residuals of our model at a parameter vector.
- **jacfn**: A function to compute the Jacobian of the residuals at a parameter vector.
- **...**: Any data needed for computation of the residual vector from the expression `rhsexpression ~ lhsvar`. Note that this is the negative of the usual residual, but the sum of squares is the same.
Details

resgr calls resfn to compute residuals and jacfn to compute the Jacobian at the parameters prm using external data in the dot arguments. It then computes the gradient using t(Jacobian) . residuals. Note that it appears awkward to use this function in calls to optimization routines. The author would like to learn why.

Value

The numeric vector with the gradient of the sum of squares at the parameters.

Author(s)

John C Nash <nashjc@uottawa.ca>

References


See Also

Function nls(), packages optim and optimx.

Examples

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[3]*zz*yy
  jj[tt,3] <- 0.01*x[1]*zz*yy*x[2]*tt
  attr(jj, "gradient") <- jj
  jj
}

st <- c(b1=1, b2=1, b3=1)
RG <- resgr(st, shobbs.res, shobbs.jac)
RG
resss

Compute sum of squares from residuals via the residual function.

Description

For a nonlinear model originally expressed as an expression of the form \( \text{lhs} \sim \text{formula}_\text{for}_\text{rhs} \) assume we have a \texttt{resfn} and \texttt{jacfn} that compute the residuals and the Jacobian at a set of parameters. This routine computes the sum of squares of the residuals.

Usage

\[
\text{resss}(\text{prm, resfn, ...})
\]

Arguments

- \texttt{prm} A parameter vector. For our example, we could use \texttt{start=c(b1=1, b2=2.345, b3=0.123)} However, the names are NOT used, only positions in the vector.
- \texttt{resfn} A function to compute the residuals of our model at a parameter vector.
- \texttt{...} Any data needed for computation of the residual vector from the expression \texttt{rhexpression - lhsvar}. Note that this is the negative of the usual residual, but the sum of squares is the same.

Details

\texttt{resss} calls \texttt{resfn} to compute residuals and then uses \texttt{crossprod} to compute the sum of squares.

At 2012-4-26 there is no checking for errors. The evaluations of residuals and the cross product could be wrapped in \texttt{try()} if the evaluation could be inadmissible.

Value

The scalar numeric value of the sum of squares at the parameters.

Author(s)

John C Nash <nashjc@uottawa.ca>

References


See Also

Function \texttt{nls()}, packages \texttt{optim} and \texttt{optimx}. 
Examples

```r
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
}

st <- c(b1=1, b2=1, b3=1)
firsts <- ress(st, shobbs.res)
  # The sum of squares of the scaled Hobbs function at parameters st
firsts
  # now illustrate how to get solution via optimization

tf <- function(prm){
  val <- ress(prm, shobbs.res)
}
testop <- optim(st, tf, control=list(trace=1))
testop
```

---

**rjfundoc**  
*Document functions that evaluate residuals at particular parameter locations.*

---

**Description**

Output a description of the model and the data that was used at the time of its creation to the console and optionally to a file. The purpose of this function is to provide a record of the details underlying the function `fun` as well as to aid users wishing to create objective functions for optimization tools.

**Usage**

```r
rjfundoc(fun, savefile=NULL)
## S3 method for class 'rjfundoc'
print(x, ...)```

**Arguments**

- `fun`  
  A function produced by `model2rjfun`.
- `savefile`  
  A character string or connection giving a location in which to record the `rjfundoc` output.
- `x, ...`  
  Object to print, and other arguments that will be ignored.
Details

rjfundoc displays the contents of the environment associated with fun.

Value

rjfundoc returns a list of class "rjfundoc" containing values extracted from the environment of fun.

Author(s)

John Nash and Duncan Murdoch

See Also

nls

Examples

```r
## require(nlsr)
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)
weeddata2 <- data.frame(y=1.5*ydat, tt=tdata)
estal <- y - ms*b1/(1+b2*exp(-b3*tt))
ms <- 1
weeddata3<-weeddata2
weeddata3$y<-0
grs3<-model2rjfun(estal, start1, data=weeddata3, ms=ms)
res3<-grs3(start1)
res3
rjfundoc(grs3)

# Now a different value of ms
ms<-2
grs3b<-model2rjfun(estal, start1, data=weeddata3, ms=ms)
res3b<-grs3b(start1)
res3b
rjfundoc(grs3b)
# rjfundoc(grs3b, savefile="grs3save.txt") ## to save the output
```
**Description**

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

**Usage**

```r
## S3 method for class 'nlsr'
summary(object, ...)
```

**Arguments**

- `object` An object of class 'nlsr'
- `...` Currently ignored.

**Details**

`summary.nlsr` performs a summary method for an object of class 'nlsr' that has been created by a routine such as nlfb or nlxb for nonlinear least squares problems.

Issue: When there are bounded parameters, nls returns a Standard Error for each of the parameters. However, this summary does NOT have a Jacobian value (it is set to 0) for columns where a parameter is masked or at (or very close to) a bound. See the R code for the determination of whether we are at a bound. In this case, users may wish to look in the `inst/dev-codes` directory of this package, where there is a script `seboundsnlsrx.R` that computes the nls() standard errors for comparison on a simple problem.

Issue: The printsum() of this object includes the singular values of the Jacobian. These are displayed, one per coefficient row, with the coefficients. However, the Jacobian singular values do NOT have a direct correspondence to the coefficients on whose display row they appear. It simply happens that there are as many Jacobian singular values as coefficients, and this is a convenient place to display them. The same issue applies to the gradient components.

**Value**

returns an invisible copy of the nlsr object.

**Author(s)**

John C Nash <nashjc@uottawa.ca>

**See Also**

Function nls(), packages optim and optimx.
wrapnlsr

Provides class nls solution to a nonlinear least squares solution using the Nash Marquardt tools.

Description

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 resulting solution is fed into the nls() function in an attempt to get the nls class solution.

Usage

wrapnlsr(formula, start, trace=FALSE, data, lower=-Inf, upper=Inf, control=list(), ...)

Arguments

- **formula**: This is a modeling formula of the form (as in nls) lhs ~ rhsexpression for example, y ~ b1/(1+b2*exp(-b3*tt)) You may also give this as a string.
- **start**: A named parameter vector. For our example, we could use start=c(b1=1, b2=2.345, b3=0.123)
- **trace**: Logical TRUE if we want intermediate progress to be reported. Default is FALSE.
- **data**: A data frame containing the data of the variables in the formula. This data may, however, be supplied directly in the parent frame.
- **lower**: Lower bounds on the parameters. If a single number, this will be applied to all parameters. Default -Inf.
- **upper**: Upper bounds on the parameters. If a single number, this will be applied to all parameters. Default Inf.
- **control**: A list of controls for the algorithm. These are as for nls().
- **...**: Any data needed for computation of the residual vector from the expression rhsexpression - lhsvar. Note that this is the negative of the usual residual, but the sum of squares is the same.

Details

wrapnlsr first attempts to solve the nonlinear sum of squares problem by using nlsMNQ, then takes the parameters from that method to call nls.

Value

An object of type nls.
Author(s)

John C Nash <nashjc@uottawa.ca>

See Also

Function nls(), packages optim and optimx.

Examples

cat("See more examples in nlmrt-package.Rd\n")
cat("kvanderpoel.R test of wrapnlsr\n")
# require(nlsr)
x<-c(1,3,5,7)
y<-c(37.98,11.68,3.65,3.93)
pks28<-data.frame(x=x,y=y)
fit0<-try(nls(y~(a+b*exp(1)^(-c*x)), data=pks28, start=c(a=0,b=1,c=1),
           trace=TRUE))
print(fit0)
cat("\n\n")
fit1<-nlxb(y~(a+b*exp(-c*x)), data=pks28, start=c(a=0,b=1,c=1), trace = TRUE)
print(fit1)
cat("\n\nor better\n")
fit2<-wrapnlsr(y~(a+b*exp(-c*x)), data=pks28, start=c(a=0,b=1,c=1),
              lower=-Inf, upper=Inf, trace = TRUE)
Index

* math
  findSubexpres, 8
  nlsDeriv, 13
  nlsSimplify, 16

* nls
  nlsr-package, 2

* nonlinear least squares
  coef.nlsr, 6
  nlfb, 11
  nlsr-package, 2
  nlxb, 18
  predict.nlsr, 20
  print.nlsr, 21
  res, 22
  resgr, 23
  resss, 25
  summary.nlsr, 28
  wrapnlsr, 29

* nonlinear
  model2rjfun, 9
  nlsDeriv, 13
  rjfundoc, 26

alist, 14

  codeDeriv(nlsDeriv), 13
  coef.nlsr, 6

  D, 14
  deriv, 9, 14, 15
  dex, 7, 15

  findSubexpres, 8
  fnDeriv, 9
  fnDeriv(nlsDeriv), 13

  isCALL(nlsSimplify), 16
  isFALSE(nlsSimplify), 16
  isMINUSONE(nlsSimplify), 16
  isONE(nlsSimplify), 16
  isTRUE, 17

  isZERO(nlsSimplify), 16

  model2rjfun, 9
  model2ssgrfun(model2rjfun), 9
  modeleexpr(model2rjfun), 9

  newDeriv(nlsDeriv), 13
  newSimplification(nlsSimplify), 16
  nlfb, 11
  nls, 10, 27
  nlsDeriv, 7, 8, 13, 18
  nlsr(nlsr-package), 2
  nlsr-package, 2
  nlsSimplify, 9, 14, 15, 16
  nlxb, 18

  optim, 7, 13, 20, 22, 24, 25, 28, 30

  predict.nlsr, 20
  print.nlsr, 21
  print.rjfundoc (rjfundoc), 26

  res, 22
  resgr, 23
  resss, 25
  rjfundoc, 26

  substitute, 14
  summary.nlsr, 28
  sysDerivs(nlsDeriv), 13
  sysSimplifications(nlsSimplify), 16

  wrapnlsr, 29