nlsr/nlsr Background, Development, and Discussion

John C. Nash

2023-09-02

Overview and objectives

This extended vignette is to explain and to (partially) record 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**. As of 2022, it also records the upgrade to **nlsr**.

A particular goal in **nlsr** (and previously **nlsr**) is to attempt, wherever possible, to use analytic or automatic derivatives. The function nls() generally uses a rather weak forward derivative approximation, though a central difference approximation is available if one uses advanced options.

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**

Two other objectives have arisen in 2022 in the move to a new **nlsr**:

- to allow for variations in the method of solving the Gauss-Newton equations, so that we may more easily test the performance of different approaches by changing control parameters to our programs. In practice, these changes have shown almost no substantive changes in performance; if a Marquardt stabilization of any sort is used, it seems to provide similar advantages over the simple Gauss-Newton approach of `nls()`.
- to employ the `roxygen2` approach to documenting routines. This choice does confer the advantage of consolidating the documentation (.Rd) files in the source code (.R) files, as well as building the NAMESPACE file more or less automatically. On the other hand, it brings some additional syntactic complications and the need to remember to `roxygenise()` the package before building and using it.

In preparing the **nlsr** package there was 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. This was expanded in 2021 with a Google Summer of Code initiative **Improvements to nls()** in which Arkajyoti Bhattacharjee was the contributor. Unfortunately, while we made some progress, we were NOT able to overcome some of the densely entangled code sufficiently to make more than limited improvements.

A large part of the work for the `nlsr` package family – particularly the parts concerning derivatives and R language structures – was initially carried out by Duncan Murdoch in 2014. Without his input, much of the capability of the package would not exist, even though there was an earlier 2012 package `nlmrt` (John C. Nash (2012)). That package was clumsily written, but did show the possibilities for automatically providing analytic Jacobian information to a nonlinear regression package.

The `nlsr` package and this vignette are works in progress, and assistance and examples are welcome. Note that there is similar work on symbolic derivatives 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. There are also aspects of `nls()` in base R and the package `minpack.lm` (Elzhov et al. (2012)) which could be better aligned. Much more on `nls()` in particular is in process with Arkajyoti Bhattacharjee in mid 2022.

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.

0. Issues remaining to address and TODOS

Issues are related to concerns about one or more of the nonlinear modeling tools as revealed by tests and examples used in this vignette. I have labeled some of these as MINOR, and regard these as issues that could be fixed easily.

nls() uses different models with different algorithm choices

In Section 7. under Partially Linear Models, `nls()` uses different model forms according to the choice of `algorithm` in some cases where the model can be partially linear. I believe there should at least be a WARNING about this possibility, though preferably I would like to see this treated as an error in the code. This is discussed in more detail in the paper "Refactoring nls()" (John C. Nash and Bhattacharjee (2022)).

MINOR – Bounds specification and warnings for nlsLM and nls.lm

`nls()`, `nlsr::nlxb` and `nlsr::nlfb` allow a single value to be given for the `lower` and `upper` bounds. This single value is expanded to a vector of length equal to the number of parameters, but the `minpack.lm` routines are more fussy.

Also, while `nls()` and the `nlsr` functions warn of initial starting parameters that violate specified bounds, the `minpack.lm` routines do not. This is easily fixable with a `warning()`.

These are illustrated below.

```
# Bounds Test nlsbtsimple.R (see BT.RES in Nash and Walker-Smith (1987))
rm(list=ls())
bt.res<-function(x){ x }
bt.jac<-function(x){nn <- length(x); JJ <- diag(nn); attr(JJ, "gradient") <- JJ;  JJ}
n <- 4
x<-rep(0,n) ; lower<-rep(NA,n); upper<-lower # to get arrays set
for (i in 1:n) { lower[i]<-1.0*(i-1)*(n-1)/n; upper[i]<-1.0*i*(n+1)/n }
x <-0.5*(lower+upper) # start on mean
xnames<-as.character(1:n) # name our parameters just in case
for (i in 1:n){ xnames[i]<-paste("p",xnames[i],sep='') }
names(x) <- xnames
require(minpack.lm)
require(nlsr)
bsnlf0<-nlfb(start=x, resfn=bt.res, jacfn=bt.jac) # unconstrained
bsnlf0
```

```
## residual sumsquares =  3.689e-16  on  4 observations
##     after  3    Jacobian and  3 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## p1           2.49965e-09         Inf           0       NaN     2.5e-09           1
## p2           6.49909e-09         Inf           0       NaN   6.499e-09           1
## p3           1.04985e-08         Inf           0       NaN    1.05e-08           1
## p4            1.4498e-08         Inf           0       NaN    1.45e-08           1
```

```
bsnlm0<-nls.lm(par=x, fn=bt.res, jac=bt.jac) # unconstrained
bsnlm0
```

```
## Nonlinear regression via the Levenberg-Marquardt algorithm
## parameter estimates: 0, 0, 0, 0
## residual sum-of-squares: 0
## reason terminated: The cosine of the angle between `fvec' and any column of the Jacobian is at most
```

```
bsnlf1<-nlfb(start=x, resfn=bt.res, jacfn=bt.jac, lower=lower, upper=upper)
bsnlf1
```

```
## residual sumsquares =  7.875  on  4 observations
##     after  6    Jacobian and  6 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## p1           4.72611e-12          NA          NA       NaN   4.726e-12           1
## p2                 0.75L          NA          NA       NaN           0           0
```

```
## p3                  1.5L           NA         NA        NaN           0           0
## p4                 2.25L           NA         NA        NaN           0           0
```

```
bsnlm1<-nls.lm(par=x, fn=bt.res, jac=bt.jac, lower=lower, upper=upper)
bsnlm1
```

```
## Nonlinear regression via the Levenberg-Marquardt algorithm
## parameter estimates: 0, 0.75, 1.5, 2.25
## residual sum-of-squares: 7.88
## reason terminated: Relative error between `par' and the solution is at most `ptol'.
```

```
# single value bounds
bsnlf2<-nlfb(start=x, resfn=bt.res, jacfn=bt.jac, lower=0.25, upper=4)
bsnlf2
```

```
## residual sumsquares =  0.25  on  4 observations
##     after  5    Jacobian and  5 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## p1                 0.25L           NA         NA        NaN           0           0
## p2                 0.25L           NA         NA        NaN           0           0
## p3                 0.25L           NA         NA        NaN           0           0
## p4                 0.25L           NA         NA        NaN           0           0
```

```
# nls.lm will NOT expand single value bounds
bsnlm2<-try(nls.lm(par=x, fn=bt.res, jac=bt.jac, lower=0.25, upper=4))
```

```
## Error in nls.lm(par = x, fn = bt.res, jac = bt.jac, lower = 0.25, upper = 4) :
##   length(lower) must be equal to length(par)
```

For example (from the file `nlsbtsimple.R`):

MINOR – nlsLM and nls.lm do not warn of out of bounds initial parameters

```
x<-rep(0,n) # resetting to this puts x out of bounds
cat("lower:"); print(lower)
```

```
## lower:
```

```
## [1] 0.00 0.75 1.50 2.25
```

```
cat("upper:"); print(upper)
```

```
## upper:
```

```
## [1] 1.25 2.50 3.75 5.00
```

```
names(x) <- xnames # to ensure names set
obnlm<-nls.lm(par=x, fn=bt.res, jac=bt.jac, lower=lower, upper=upper)
obnlm
```

```
## Nonlinear regression via the Levenberg-Marquardt algorithm
## parameter estimates: 0, 0.75, 1.5, 2.25
## residual sum-of-squares: 7.88
## reason terminated: Relative error between `par' and the solution is at most `ptol'.
```

For nonlinear regression with `minpack.lm::nlsLM` there is also no warning. The example using file `Hobbsbdformula1.R` in the vignette **R: Examples of different nonlinear least squares calculations** (file `NLS-Examples.Rmd`) shows this. However, in this case, the program actually gets a good answer, despite the failure to warn. In a start from all 1's, which is feasible, a poor answer is obtained, as noted in the next section.

Bounded minimization with minpack.lm

While minpack.lm mentions lower and upper bounds in the manual pages, there are no examples in the package (that I could find) of their application. For the file `nlsbtsimple.R`, we get acceptable answers. For the Hobbs problem, from a start of all 1's, both `nlsLM` and `nls.lm` converge to sums of squares higher than `nlxb`, `nlfb` or `nls()` using the "port" algorithm (when the last is able to get started). Moreover, in one case for the scaled Hobbs problem, the two `minpack.lm` functions give different results. I have not yet been able to understand how these routines apply bounds constraints. There is mention in https://lmfit-py.readthedocs.io/en/latest/bounds.html that the original MINPACK did NOT cater for bounds constraints on parameters, and that MINUIT, which uses the MINPACK ideas, used a transformation of the parameters to accomplish this. (Hans Werner Borchers uses the same idea in the the code `dfoptim::nmkb`, which he calls a **transfinite transformation**.)

Comments:

1) the transfinite idea is useful in that it may be applicable quite generally. If a wrapper for unconstrained minimizers could be devised, it would enlarge the capability of a number of R tools.

2) The Hobbs problem, even when scaled, may be unscaled again by such a transformation of the parameters.

I have seen many cases where methods that are generally reliable give an occasional unsatisfactory result. However, it would be useful to know the precise reasons why or where `minpack.lm` routines are obtaining these results, since it may be possible to either fix the issues or else provide some warnings or diagnostics, which hopefully would be of wider application.

TODOS

- sysDerivs and sysSimplifications are new environments whose parent is emptyenv(). Why? This should be better explained with motivations.
- R function `optim()` and function `optimr()` in package `optimx` include the control `parscale` which is a vector of scaling factors so that optimization is performed on `par/parscale` where `par` are the user-supplied parameters. The intent is to have the internal scaled parameters of similar general size. The `Hobbssetup` script below carries out this scaling explicitly. Providing a `parscale` capability for `nlxb()` and `nlfb()` is mostly a matter of effort. At the time of writing (August 2022), I feel that the performance of the functions is good and adding `parscale` is not an urgent need.
- `nls()` does not insist that the `formula` argument in its call is of the structure where the left hand side is "simple", that is, usually a single variable. However, it is difficult to find examples where this is not the case, though it is NOT a requirement of the nonlinear least squares algorithms. However, to get the right features for more complicated modelling formulas, I need collaboration with practitioners.
- Data can be passed to `nls()` using variables already in the working environment or in the `data` argument, which can be a dataframe, a list or an environment, but not a matrix. Why can a matrix of data not be used?
- How can VARPRO / conditional linearity be married to the nlsr algorithms?
- A long-term need is a better, more consistent way to specify formulas for nonlinear models. At the moment `nlsr` does not support indexed parameters, and `nls()` does so in an awkward way. The output parameters do not match the input specification in that they are not indexed using the traditional square brackets, but build the index value into the parameter name.

1. Summary of capabilities and functions in `nlsr`

Throughout this exposition, we have chosen to set `trace` variables to `FALSE` to reduce the volume of output. Changing such values to `TRUE` will expand output.

We will illustrate many of the capabilities with the Hobbs weed problem (J. C. Nash (1979), page 121). This can be set up in a scaled or unscaled form.

```
## Use the Hobbs Weed problem
weed <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443,
          38.558, 50.156, 62.948, 75.995, 91.972)
tt <- 1:12
weeddf <- data.frame(y=weed, tt=tt)
st <- c(b1=1, b2=1, b3=1) # a default starting vector (named!)
## Unscaled model
wmodu <- y ~ b1/(1+b2*exp(-b3*tt))
## Scaled model
wmods <- y ~ 100*b1/(1+10*b2*exp(-0.1*b3*tt))
## We can provide the residual and Jacobian as functions
# Unscaled Hobbs problem
hobbs.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  <-  x[1]/(1+x[2]*exp(-x[3]*tt)) - y
}

hobbs.jac  <-  function(x) { # scaled Hobbs weeds problem -- Jacobian
  jj  <-  matrix(0.0, 12, 3)
  tt  <-  1:12
  yy  <-  exp(-x[3]*tt)
  zz  <-  1.0/(1+x[2]*yy)
  jj[tt,1]   <-   zz
  jj[tt,2]   <-   -x[1]*zz*zz*yy
  jj[tt,3]   <-   x[1]*zz*zz*yy*x[2]*tt
  attr(jj, "gradient") <- jj
  jj
}
# Scaled Hobbs problem
shobbs.res  <-  function(x){ # scaled Hobbs weeds problem -- residual
  # This variant uses looping
  if(length(x) != 3) stop("shobbs.res -- parameter vector n!=3")
  y  <-  c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443,
           38.558, 50.156, 62.948, 75.995, 91.972)
  tt  <-  1:12
  res  <-  100.0*x[1]/(1+x[2]*10.*exp(-0.1*x[3]*tt)) - y
}

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

```
  jj
}
```

nlxb

This is the likely the function most called by users from `nlsr`.

- 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 as a default method 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`.

```
require(nlsr)
# Use Hobbs scaled formula model
anlxb1  <-  try(nlxb(wmods, start=st, data=weeddf))
print(anlxb1)
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131       17.35  3.167e-08   7.809e-08       130.1
## b2               4.90916        0.1688       29.08  3.284e-10   3.578e-07       6.165
## b3                3.1357       0.06863       45.69  5.768e-12  -3.459e-07       2.735
```

```
# summary(anlxb1) # for different output
# Test without starting parameters
anlxb1nostart  <-  try(nlxb(wmodu, data=weeddf))
```

```
## Error in nlxb(wmodu, data = weeddf) : A start vector MUST be provided
```

```
print(anlxb1nostart)
```

```
## [1] "Error in nlxb(wmodu, data = weeddf) : A start vector MUST be provided\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in nlxb(wmodu, data = weeddf): A start vector MUST be provided>
```

Here we see that the Jacobian at the solution is essentially of rank 1, even though there are 3 coefficients. It is therefore not surprising that `nls()`, which does not benefit from the Levenberg-Marquardt stabilization in solving the nonlinear least squares program fails for this problem. See the subsection below for `wrapnlsr`. Note that the singular values of the Jacobian computed via a numerical approximation are more extreme (i.e., nearly singular) than those determined by analytic derivatives in the preceding solution `anlxb1`. For this example, there is a clear advantage to the analytic derivatives.

While there is an almost liturgical adherence to setting up models where the dependent (or predicted) variable is on the left hand side (LHS) of the model formula and the independent (or predictor) variable(s) on the right hand side (RHS), this is
NOT an actual requirement in most cases, though there are some situations such as the `minpack.lm` function `wfct` that do assume this structure. Here is an example of solving the Hobbs unscaled problem using a 1-sided model formula.

```
# One-sided unscaled Hobbs weed model formula
wmodu1  <-  ~ b1/(1+b2*exp(-b3*tt)) - y
anlxb11  <-  try(nlxb(wmodu1, start=st, data=weeddf))
print(anlxb11)
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  19    Jacobian and  25 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               196.186         11.31      17.35  3.167e-08  -4.859e-09        1011
## b2               49.0916         1.688      29.08  3.284e-10  -3.099e-08      0.4605
## b3               0.31357      0.006863      45.69  5.768e-12   2.305e-06     0.04714
```

model2rjfun, model2ssgrfun, modelexpr

- These functions create functions to evaluate residuals or sums of squares at particular parameter locations. `model2rjfun` is the key call in `nlxb()` to estimate models specified as expressions or formulas.

```
# st  <-  c(b1=1, b2=1, b3=1)
wrj <- model2rjfun(wmods, st, data=weeddf)
wrj
```

```
## function (prm)
## {
##     if (is.null(names(prm)))
##         names(prm) <- names(pvec)
##     localdata <- list2env(as.list(prm), parent = data)
##     eval(residexpr, envir = localdata)
## }
## <bytecode: 0x5598782ea590>
## <environment: 0x559874f87d40>
```

```
weedux <- nlxb(wmods, start=st, data=weeddf)
print(weedux)
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08   7.809e-08       130.1
## b2               4.90916        0.1688      29.08  3.284e-10   3.578e-07       6.165
## b3                3.1357       0.06863      45.69  5.768e-12  -3.459e-07       2.735
```

```
wss <- model2ssgrfun(wmods, st, data=weeddf)
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: 0x559874318dc0>
## <environment: 0x5598742fcea0>
```

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

```
## expression({
##     .expr1 <- 100 * b1
##     .expr2 <- 10 * b2
##     .expr6 <- exp(-0.1 * b3 * tt)
##     .expr8 <- 1 + .expr2 * .expr6
##     .expr14 <- .expr8^2
##     .value <- .expr1/.expr8 - y
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("b1",
##         "b2", "b3")))
##     .grad[, "b1"] <- 100/.expr8
##     .grad[, "b2"] <- -(.expr1 * (10 * .expr6)/.expr14)
##     .grad[, "b3"] <- .expr1 * (.expr2 * (.expr6 * (0.1 * tt)))/.expr14
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

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

```
## expression({
##     .expr1 <- 100 * b1
##     .expr2 <- 10 * b2
##     .expr6 <- exp(-0.1 * b3 * tt)
##     .expr8 <- 1 + .expr2 * .expr6
##     .expr14 <- .expr8^2
##     .value <- .expr1/.expr8 - y
##     .grad <- array(0, c(length(.value), 3L), list(NULL, c("b1",
##         "b2", "b3")))
##     .grad[, "b1"] <- 100/.expr8
##     .grad[, "b2"] <- -(.expr1 * (10 * .expr6)/.expr14)
##     .grad[, "b3"] <- .expr1 * (.expr2 * (.expr6 * (0.1 * tt)))/.expr14
##     attr(.value, "gradient") <- .grad
##     .value
## })
```

nlfb – minimize nonlinear least squares residual functions

- 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 `jacfn` is missing, then a numerical approximation to derivatives can be used if the control `japprox` is appropriately specified. Valid choices for approximations are `jafwd`, `jaback`, `jacentral` and `jand` for forward, backward, central and package `numDeriv` difference methods. There is also the choice `SSJac`, which is not necessarily an approximation, but gradient code within a selfStart model function. (CAUTION: There is no check that such code is actually present!) The Jacobian is stored in the attribute `gradient` of the residual allowing us to combine the computation of the residual and Jacobian in the same code. That is, we can specify the same R function for both `resfn` and `jacfn`. Since there are generally common computations, this may give a small improvement in efficiency, though it does make the setup slightly more complicated. See the example below. However,

the main reason for this choice was to allow `nlxb` to be more easily coded.

- 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 John C. Nash (1977), J. C. 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.

```
require(nlsr)

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

```
## try nlfb
```

```
st  <-  c(b1=1, b2=1, b3=1)
ans1 <- nlfb(st, shobbs.res, shobbs.jac, trace=FALSE)
summary(ans1)
```

```
## Object has try-error or missing parameters
```

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

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08   6.669e-08       130.1
## b2               4.90916        0.1688      29.08  3.284e-10   3.232e-07       6.165
## b3                3.1357       0.06863      45.69  5.768e-12  -3.805e-07       2.735
```

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

```
##          b1          b2          b3
## -2.3909e-10  1.2708e-09  4.4685e-10
## attr(,"pkgname")
## [1] "nlsr"
```

coef.nlsr

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

```
coef(anlxb1) # this is solution of scaled problem from unit start
```

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

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
```

```
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08   7.809e-08       130.1
## b2               4.90916        0.1688      29.08  3.284e-10   3.578e-07       6.165
## b3                3.1357       0.06863      45.69  5.768e-12  -3.459e-07       2.735
```

```
print(ans1)
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08   7.809e-08       130.1
## b2               4.90916        0.1688      29.08  3.284e-10   3.578e-07       6.165
## b3                3.1357       0.06863      45.69  5.768e-12  -3.459e-07       2.735
```

summary.nlsr

- Provide a summary output (but involving some serious computations!) of an object of class nlsr from nlxb or nlfb from package nlsr. Also available for nls(). Note that this gives a rather lengthy output, so calling it a summary is a bit of a stretch.

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

```
## Object has try-error or missing parameters
```

```
summary(ans1)
```

```
## Object has try-error or missing parameters
```

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 J. C. Nash (1979) of the Marquardt algorithm, where the linear sub-problem is solved by a qr method.

A particular purpose of this function is to create the `nls`-style model object for a problem when the solution has been obtained by `nlsr::nlxb`. `minpack.lm::nlsLM` creates a structure that is parallel to that from `nls()`.

```
## The following attempt at Hobbs unscaled with nls() fails!
rm(list=ls()) # clear before starting
weed <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443,
          38.558, 50.156, 62.948, 75.995, 91.972)
tt <- 1:12
weeddf <- data.frame(tt, weed)
st <- c(b1=1, b2=1, b3=1) # a default starting vector (named!)
wmodu<- weed ~ b1/(1 + b2 * exp(- b3 * tt))
anls1u  <-  try(nls(wmodu, start=st, trace=FALSE, data=weeddf)) # fails
```

```
## Error in nls(wmodu, start = st, trace = FALSE, data = weeddf) :
##   singular gradient
```

```
## But we succeed by calling nlxb first.
library(nlsr)
anlxb1un  <-  try(nlxb(wmodu, start=st, trace=FALSE, data=weeddf))
print(anlxb1un)
```

```
## residual sumsquares =  2.5873  on  12 observations
```

```
##     after  19    Jacobian and  25 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               196.186         11.31      17.35  3.167e-08  -4.859e-09        1011
## b2               49.0916         1.688      29.08  3.284e-10  -3.099e-08      0.4605
## b3               0.31357      0.006863      45.69  5.768e-12   2.305e-06     0.04714
```

```
st2 <- coef(anlxb1un) # We try to start nls from solution using nlxb
anls2u  <-  try(nls(wmodu, start=st2, trace=FALSE, data=weeddf))
print(anls2u)
```

```
## Nonlinear regression model
##   model: weed ~ b1/(1 + b2 * exp(-b3 * tt))
##    data: weeddf
##      b1      b2      b3
## 196.186  49.092   0.314
##  residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 4.3e-08
```

```
## Or we can simply call wrapnlsr FROM THE ORIGINAL start
anls2a  <-  try(wrapnlsr(wmodu, start=st, trace=FALSE, data=weeddf))
summary(anls2a)
```

```
##
## Formula: weed ~ b1/(1 + b2 * exp(-b3 * tt))
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## b1 1.96e+02   1.13e+01    17.4  3.2e-08 ***
## b2 4.91e+01   1.69e+00    29.1  3.3e-10 ***
## b3 3.14e-01   6.86e-03    45.7  5.8e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.536 on 9 degrees of freedom
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 2.03e-08
```

```
# # For comparison could run nlsLM
# require(minpack.lm)
# anlsLM1u  <-  try(nlsLM(wmodu, start=st, trace=FALSE, data=weeddf))
# print(anlsLM1u)
```

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 -- Scaled Hobbs problem
shobbs.res  <-  function(x){ # scaled Hobbs weeds problem -- residual
  if(length(x) != 3) stop("shobbs.res -- parameter vector n!=3")
  y  <-  c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443,
           38.558, 50.156, 62.948, 75.995, 91.972)
  tt  <-  1:12
```

```
  res  <-  100.0*x[1]/(1+x[2]*10.*exp(-0.1*x[3]*tt)) - y
}

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

```
## jacfn: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
## }
## <bytecode: 0x55987a207e70>
```

```
RG
```

```
##  [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(,"Jacobian")
##          [,1]     [,2]     [,3]
##  [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
## attr(,"Jacobian")attr(,"gradient")
##          [,1]     [,2]     [,3]
##  [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
## attr(,"gradient")
## [1] -10091.3   7835.3  -8234.2
```

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

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

nlsDeriv, codeDeriv, fnDeriv, newDeriv

These functions are mainly for test and development use.

- 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.

```
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, though joe() is meaningless.
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.
```

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: 0x559877ef9d28>
```

```
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?
# Where are derivative definitions are stored?
str(sysDerivs)
```

```
## <environment: 0x559879026b80>
```

2. Analytic versus approximate Jacobians

A key goal of package `nlsr` was to be able to use analytic or symbolic derivatives for the Jacobian in nonlinear least squares computations, in particular when the model is specified as a formula or expression. In this `nlsr::nlxb()` has been quite successsful. It should be pointed out that the principal advantage of using analytic derivatives is that we get a more assured measure of the "flatness" of the sum of squares surface

at the termination point. My experience is that there is no particular gain in the speed in getting to that termination point.

A disadvantage of the approach is that specifying a "formula" that includes an R function will (usually) fail. `nls()`, because it defaults to using a derivative approximation, can accept formulas that include R functions, and indeed, one of the examples in the manual page `nls.Rd` is of this type. In package `nlsr` we have introduced the possibility of using Jacobian approximations via the control element `japprox` which is discussed in several places below.

A second goal of including approximations for the Jacobian is to be able to specify or control the approximation. `nls()`, as shown in Appendix A, has a rather complicated code involving both R and C to compute a forward difference approximation to the Jacobian. In this computation, each parameter is adjusted by its absolute value times the square root of the machine precision (double). Call this the `ndstep`. So the parameter `delta` is the absolute value of the parameter times approximately 1.5e-8. If the parameter is zero, then the `delta` is `ndstep`. In `nlsr::jafwd()`, I use a `delta` of `ndstep` times the absolute value of the parameter PLUS the `ndstep`. This avoids the check for a zero parameter.

It is known (https://en.wikipedia.org/wiki/Finite_difference) that the forward (and backward) difference approximations are not ideal. Central differences and higher approximations such as those found in the CRAN package `numDeriv` are better, and it is desirable to be able to specify that these be used.

Specifying approximations to nlfb

We first look at the `nlfb()` function that uses R functions for the residual and Jacobian. Borrowing from the mechanism used in `optimx::optimr()`, we can invoke an approximation by putting the name of an appropriate R function in quotation marks. In `nlsr` there are four functions `jafwd()`, `jaback`, `jacentral`, and `jand` for the forward, backward, central and `numDeriv` (default) approximations. Moreover, the `ndstep` can be set in the `control()` list in the `nlfb()` call. Its default value is 1e-7. An open question is whether to change to the value `sqrt(.Machine$double.eps)` to more closely match `nls()`.

```
## Test with functional spec. of problem
## Default call WITH jacobian function
ans1 <- nlfb(st, resfn=shobbs.res, jacfn=shobbs.jac)
ans1
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08   7.809e-08       130.1
## b2               4.90916        0.1688      29.08  3.284e-10   3.578e-07       6.165
## b3                3.1357       0.06863      45.69  5.768e-12  -3.459e-07       2.735
```

```
## No jacobian function -- and no japprox control setting
ans1n <- try(nlfb(st, shobbs.res)) # NO jacfn
```

```
## Error in nlfb(st, shobbs.res) :
##   MUST PROVIDE jacobian function or specify approximation
```

```
ans1n
```

```
## [1] "Error in nlfb(st, shobbs.res) : \n  MUST PROVIDE jacobian function or specify approximation\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in nlfb(st, shobbs.res): MUST PROVIDE jacobian function or specify approximation>
```

```
## Force jafwd approximation
ans1nf <- nlfb(st, shobbs.res, control=list(japprox="jafwd")) # NO jacfn, but specify fwd approx
ans1nf
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131       17.35  3.167e-08   6.669e-08       130.1
## b2               4.90916        0.1688       29.08  3.284e-10   3.232e-07       6.165
## b3                3.1357       0.06863       45.69  5.768e-12  -3.805e-07       2.735
```

```
## Alternative specification
ans1nfa <- nlfb(st, shobbs.res, jacfn="jafwd") # NO jacfn, but specify fwd approx in jacfn
ans1nfa
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131       17.35  3.167e-08   6.669e-08       130.1
## b2               4.90916        0.1688       29.08  3.284e-10   3.232e-07       6.165
## b3                3.1357       0.06863       45.69  5.768e-12  -3.805e-07       2.735
```

```
## Coeff differences from analytic:
ans1nf$coefficients-ans1$coefficients
```

```
##          b1          b2          b3
##  2.3909e-10 -1.2708e-09 -4.4685e-10
```

```
## Force jacentral approximation
ans1nc <- nlfb(st, shobbs.res, control=list(japprox="jacentral")) # NO jacfn
ans1nc
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131       17.35  3.167e-08   9.537e-08       130.1
## b2               4.90916        0.1688       29.08  3.284e-10   3.587e-07       6.165
## b3                3.1357       0.06863       45.69  5.768e-12  -3.708e-07       2.735
```

```
## Coeff differences from analytic:
ans1nc$coefficients-ans1$coefficients
```

```
##          b1          b2          b3
## -5.6010e-10 -8.5208e-10  2.6850e-10
```

```
## Force jaback approximation
ans1nb <- nlfb(st, shobbs.res, control=list(japprox="jaback")) # NO jacfn
ans1nb
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131       17.35  3.167e-08   1.044e-07       130.1
## b2               4.90916        0.1688       29.08  3.284e-10   3.572e-07       6.165
## b3                3.1357       0.06863       45.69  5.768e-12  -3.119e-07       2.735
```

```
## Coeff differences from analytic:
ans1nb$coefficients-ans1$coefficients
```

```
##          b1          b2          b3
##  6.5124e-10  1.3754e-09 -2.3150e-10
```

```
## Force jand approximation
ans1nn <- nlfb(st, shobbs.res, control=list(japprox="jand"), trace=FALSE) # NO jacfn
```

```
ans1nn
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08   7.807e-08       130.1
## b2               4.90916        0.1688      29.08  3.284e-10   3.577e-07       6.165
## b3                3.1357       0.06863      45.69  5.768e-12  -3.459e-07       2.735
```

```
## Coeff differences from analytic:
ans1nc$coefficients-ans1$coefficients
```

```
##          b1          b2          b3
## -5.6010e-10 -8.5208e-10  2.6850e-10
```

Specifying Jacobian approximations to nlxb

Since `nlxb()` provides the model as a formula or expression, we need to tell it how to get an approximate Jacobian. First, we can specify WHICH approximation to use by putting the name of the jacobian approximating function in the control list element `japprox` in quotation marks.

The default mechanism for using `nlxb()` is to create a function `trjfn` (by calling `model2rjfun()`). To make our work a lot easier, `trjfn` is used as BOTH the residual and Jacobian function by copying the created Jacobian matrix into the "gradient" attribute of the returned object from `trjfn`.

NOTE: This requirement that the Jacobian matrix be returned in the "gradient" attribute of the returned object of the `jacfn` specified in the call to `nlfb()` is one that users should be careful to observe.

When we specify a Jacobian approximation (via `control$japprox`) to `nlxb()`, the call to `model2rjfun` is made with `jacobian=FALSE` and the appropriate function for the approximation is supplied in the subsequent call the `nlfb()` to minimize the sum of squared objective. The `jacobian` parameter in the call to `model2rjfun()` defaults to TRUE.

```
## rm(list=ls()) # clear workspace for each major section
## Use the Hobbs Weed problem
weed <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443,
          38.558, 50.156, 62.948, 75.995, 91.972)
tt <- 1:12
weeddf <- data.frame(y=weed, tt=tt)
st1 <- c(b1=1, b2=1, b3=1) # a default starting vector (named!)
wmods <- y ~ 100*b1/(1+10*b2*exp(-0.1*b3*tt)) ## Scaled model
print(wmods)
```

```
## y ~ 100 * b1/(1 + 10 * b2 * exp(-0.1 * b3 * tt))
```

```
anlxb1  <-  try(nlxb(wmods, start=st1, data=weeddf))
print(anlxb1)
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08   7.809e-08       130.1
## b2               4.90916        0.1688      29.08  3.284e-10   3.578e-07       6.165
## b3                3.1357       0.06863      45.69  5.768e-12  -3.459e-07       2.735
```

```
anlxb1fwd  <-  (nlxb(wmods, start=st1, data=weeddf,
    control=list(japprox="jafwd")))
print(anlxb1fwd)
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08   6.669e-08       130.1
## b2               4.90916        0.1688      29.08  3.284e-10   3.232e-07       6.165
## b3                3.1357       0.06863      45.69  5.768e-12  -3.805e-07       2.735
```

```
## fwd - analytic
print(anlxb1fwd$coefficients - anlxb1$coefficients)
```

```
##          b1          b2          b3
##  2.3909e-10 -1.2708e-09 -4.4685e-10
```

```
anlxb1bak  <- (nlxb(wmods, start=st1, data=weeddf,
    control=list(japprox="jaback")))
print(anlxb1bak)
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08   1.044e-07       130.1
## b2               4.90916        0.1688      29.08  3.284e-10   3.572e-07       6.165
## b3                3.1357       0.06863      45.69  5.768e-12  -3.119e-07       2.735
```

```
## back - analytic
print(anlxb1bak$coefficients - anlxb1$coefficients)
```

```
##          b1          b2          b3
##  6.5124e-10  1.3754e-09 -2.3150e-10
```

```
anlxb1cen  <- (nlxb(wmods, start=st1, data=weeddf,
    control=list(japprox="jacentral")))
print(anlxb1cen)
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08   9.537e-08       130.1
## b2               4.90916        0.1688      29.08  3.284e-10   3.587e-07       6.165
## b3                3.1357       0.06863      45.69  5.768e-12  -3.708e-07       2.735
```

```
## central - analytic
print(anlxb1cen$coefficients - anlxb1$coefficients)
```

```
##          b1          b2          b3
## -5.6010e-10 -8.5208e-10  2.6850e-10
```

```
anlxb1nd  <-  (nlxb(wmods, start=st1, data=weeddf,
    control=list(japprox="jand")))
print(anlxb1nd)
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  34 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               1.96186        0.1131      17.35  3.167e-08   7.807e-08       130.1
## b2               4.90916        0.1688      29.08  3.284e-10   3.577e-07       6.165
## b3                3.1357       0.06863      45.69  5.768e-12  -3.459e-07       2.735
```

```
## numDeriv - analytic
print(anlxb1nd$coefficients - anlxb1$coefficients)
```

```
##          b1          b2          b3
## -5.7647e-12 -1.0112e-11  2.4873e-12
```

3. Weighted nonlinear regression

While the standard approach to nonlinear regression is to minimize the sum of squared residuals, there are frequently good reasons to **weight** each residual to scale them according to their importance. That is, we wish to favour those residuals believed to be more certain. Until 2020, `nlsr` did not provide for weights that could alter with the parameters of the model. That is, the weights were pre-specified, at least in `nlxb`. The `resfn` and `jacfn` of `nlfb` could, of course be specified with almost any loss function that results in a sum of squared residuals.

With the addition of the possibility of forcing the use of an approximation to the Jacobian (Section 2 above), formulas that allow the inclusion of functions (i.e., subprograms) are possible, and these may include weighting.

The following examples illustrate how this works.

Static weights

```
## weighted nonlinear regression using inverse squared variance of the response
require(minpack.lm)
Treated <- Puromycin[Puromycin$state == "treated", ]
# We want the variance of each "group" of the rate variable
# for which the conc variable is the same. We first find
# this variance by group using the tapply() function.
var.Treated <- tapply(Treated$rate, Treated$conc, var)
var.Treated <- rep(var.Treated, each = 2)
Pur.wt1nls <- nls(rate ~ (Vm * conc)/(K + conc), data = Treated,
               start = list(Vm = 200, K = 0.1), weights = 1/var.Treated^2)
Pur.wt1nlm <- nlsLM(rate ~ (Vm * conc)/(K + conc), data = Treated,
               start = list(Vm = 200, K = 0.1), weights = 1/var.Treated^2)
Pur.wt1nlx <- nlxb(rate ~ (Vm * conc)/(K + conc), data = Treated,
               start = list(Vm = 200, K = 0.1), weights = 1/var.Treated^2)
rnls <- summary(Pur.wt1nls)$residuals
ssrnls<-as.numeric(crossprod(rnls))
rnlm <- summary(Pur.wt1nlm)$residuals
ssrnlm<-as.numeric(crossprod(rnlm))
rnlx <- Pur.wt1nlx$resid
ssrnlx<-as.numeric(crossprod(rnlx))
cat("Compare nls and nlsLM: ", all.equal(coef(Pur.wt1nls), coef(Pur.wt1nlm)),"\n")
```

```
## Compare nls and nlsLM:  TRUE
```

```
cat("Compare nls and nlsLM: ", all.equal(coef(Pur.wt1nls), coef(Pur.wt1nlx)),"\n")
```

```
## Compare nls and nlsLM:  Attributes: < names for current but not for target > Attributes: < Length mis
```

```
cat("Sumsquares nls - nlsLM: ", ssrnls-ssrnlm,"\n")
```

```
## Sumsquares nls - nlsLM:  3.3862e-15
```

```
cat("Sumsquares nls - nlxb: ", ssrnls-ssrnlx,"\n")
```

```
## Sumsquares nls - nlxb:  1.0027e-12
```

Dynamic weights via the wfct() function

`minpack.lm` provides an interesting function that allows us to access current values of various quantities associated with our model. There are five possibilities, of which two are static weightings – the name of the response or the predictor variable. (It seems `wfct` assumes only one such variable.) The other three possibilities are the current values of the "fitted" model, the residuals as specified by "resid", or the "error", which is the square root of the variance of the response variable. The last possibility requires repetitions of the independent or predictor variable.

Concerns: I have found that the structure of `wfct` means that examples using it in calls to `nlsLM`, `nls` or `nlxb` with fail if they are accessed via `source()` or if the call is surrounded by a `print()` or `try()`. This remains to be sorted out.

Note that `nlxb` cannot use `fitted` or `resid` in the `wfct` function to specify weights. `nls()` generates such **functions** as part of the returned object. `nlsr` does have `predict.nlsr()` that could be used to generate fits and by extension, residuals. What is possible??

```
## minpack.lm::wfct
### Examples from 'nls' doc where wfct used ###
## Weighting by inverse of response 1/y_i:
wtt1nlm<-nlsLM(rate ~ Vm * conc/(K + conc), data = Treated,
               start = c(Vm = 200, K = 0.05), weights = wfct(1/rate))
print(wtt1nlm)

wtt1nls<-nls(rate ~ Vm * conc/(K + conc), data = Treated,
          start = c(Vm = 200, K = 0.05), weights = wfct(1/rate))
print(wtt1nls)

wtt1nlx<-nlxb(rate ~ Vm * conc/(K + conc), data = Treated,
          start = c(Vm = 200, K = 0.05), weights = wfct(1/rate))
print(wtt1nlx)

## Weighting by square root of predictor \sqrt{x_i}:
# ?? why does try() not work
wtt2nlm<-nlsLM(rate ~ Vm * conc/(K + conc), data = Treated,
       start = c(Vm = 200, K = 0.05), weights = wfct(sqrt(conc)))
print(wtt2nlm)

wtt2nls<-nls(rate ~ Vm * conc/(K + conc), data = Treated,
       start = c(Vm = 200, K = 0.05), weights = wfct(sqrt(conc)))
print(wtt2nls)

wtt2nlx<-nlxb(rate ~ Vm * conc/(K + conc), data = Treated,
       start = c(Vm = 200, K = 0.05), weights = wfct(sqrt(conc)))
print(wtt2nlx)

## Weighting by inverse square of fitted values 1/\hat{y_i}^2:
wtt3nlm<-nlsLM(rate ~ Vm * conc/(K + conc), data = Treated,
       start = c(Vm = 200, K = 0.05), weights = wfct(1/fitted^2))
print(wtt3nlm)

wtt3nls<-nls(rate ~ Vm * conc/(K + conc), data = Treated,
       start = c(Vm = 200, K = 0.05), weights = wfct(1/fitted^2))
print(wtt3nls)
```

```
# These don't work -- there is no fitted() function available in nlsr
# wtt3nlx<-try(nlxb(rate ~ Vm * conc/(K + conc), data = Treated,
#        start = c(Vm = 200, K = 0.05), weights = wfct(1/fitted^2)))
##  (nlxb(rate ~ Vm * conc/(K + conc), data = Treated,
##        start = c(Vm = 200, K = 0.05), weights = wfct(1/(resid+rate)^2)))
## (wrapnlsr(rate ~ Vm * conc/(K + conc), data = Treated,
##        start = c(Vm = 200, K = 0.05), weights = wfct(1/(resid+rate)^2)))

## Weighting by inverse variance 1/\sigma{y_i}^2:
wtt4nlm<-nlsLM(rate ~ Vm * conc/(K + conc), data = Treated,
       start = c(Vm = 200, K = 0.05), weights = wfct(1/error^2))
print(wtt4nlm)

wtt4nls<-nls(rate ~ Vm * conc/(K + conc), data = Treated,
       start = c(Vm = 200, K = 0.05), weights = wfct(1/error^2))
print(wtt4nls)

wtt4nlx<-nlxb(rate ~ Vm * conc/(K + conc), data = Treated,
       start = c(Vm = 200, K = 0.05), weights = wfct(1/error^2))
print(wtt4nlx)

wtt5nls<-nls(rate ~ Vm * conc/(K + conc), data = Treated,
             start = c(Vm = 200, K = 0.05), weights = wfct(1/resid^2))
print(wtt5nls)
wtt5nlm<-nlsLM(rate ~ Vm * conc/(K + conc), data = Treated,
             start = c(Vm = 200, K = 0.05), weights = wfct(1/resid^2))
print(wtt5nlm)
## Won't work! No resid function available from nlxb.
## wtt5nlx<-nlxb(rate ~ Vm * conc/(K + conc), data = Treated,
##             start = c(Vm = 200, K = 0.05), weights = wfct(1/resid^2))
## print(wtt5nlx)
```

Weights built into a one-sided model function

In all three formula-based nonlinear model estimators (`nls`, `nlsLM`, `nlxb`), the "formula" can be set so there is no so-called Left Hand Side (LHS).

```
st<-c(b1=1, b2=1, b3=1)
frm <- ~ b1/(1+b2*exp(-b3*tt))-y
nolhs <- nlxb(formula=frm, data=weeddf, start=st)
nolhs
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  19    Jacobian and  25 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1              196.186         11.31       17.35  3.167e-08  -4.859e-09        1011
## b2              49.0916         1.688       29.08  3.284e-10  -3.099e-08      0.4605
## b3              0.31357      0.006863       45.69  5.768e-12   2.305e-06     0.04714
```

Since we can set up problems in this way, we could, in some cases, build weights into the expression on the right hand side. For `nlxb`, attempts to develop the analytic Jacobian will generally fail if the formula includes an R function call. This can be overcome by specifying a Jacobian approximation in the `control` list element `japprox`.

```r
## weighted nonlinear regression using 1-sided model functions
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)
}

Pur.wtMMnls <- nls( ~ weighted.MM(rate, conc, Vm, K), data = Treated,
               start = list(Vm = 200, K = 0.1))
print(Pur.wtMMnls)
```

```
## Nonlinear regression model
##   model: 0 ~ weighted.MM(rate, conc, Vm, K)
##    data: Treated
##       Vm        K
## 206.8347   0.0546
##  residual sum-of-squares: 14.6
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 3.86e-06
```

```r
Pur.wtMMnlm <- nlsLM( ~ weighted.MM(rate, conc, Vm, K), data = Treated,
                   start = list(Vm = 200, K = 0.1))
print(Pur.wtMMnlm)
```

```
## Nonlinear regression model
##   model: 0 ~ weighted.MM(rate, conc, Vm, K)
##    data: Treated
##       Vm        K
## 206.8347   0.0546
##  residual sum-of-squares: 14.6
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 1.49e-08
```

```r
Pur.wtMMnlx <- nlxb( ~ weighted.MM(rate, conc, Vm, K), data = Treated,
                   start = list(Vm = 200, K = 0.1), control=list(japprox="jafwd"))
print(Pur.wtMMnlx)
```

```
## residual sumsquares =  14.597  on  12 observations
##     after  8    Jacobian and  8 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## Vm               206.835         9.225      22.42  7.003e-10  -5.246e-10       230.7
## K              0.0546112      0.007979      6.845  4.488e-05  -6.713e-06       0.131
```

```r
# Another approach
## Passing arguments using a list that can not be coerced to a data.frame
lisTreat <- with(Treated, list(conc1 = conc[1], conc.1 = conc[-1], rate = rate))
```

```
weighted.MM1 <- function(resp, conc1, conc.1, Vm, K){
  conc <- c(conc1, conc.1)
  pred <- (Vm * conc)/(K + conc)
  (resp - pred) / sqrt(pred)
}
Pur.wtMM1nls <- nls( ~ weighted.MM1(rate, conc1, conc.1, Vm, K),
                data = lisTreat, start = list(Vm = 200, K = 0.1))
print(Pur.wtMM1nls)
```

```
## Nonlinear regression model
##   model: 0 ~ weighted.MM1(rate, conc1, conc.1, Vm, K)
##    data: lisTreat
##       Vm        K
## 206.8347   0.0546
##  residual sum-of-squares: 14.6
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 3.86e-06
```

```
# Should we put in comparison of coeffs / sumsquares??
# stopifnot(all.equal(coef(Pur.wt), coef(Pur.wt1)))
Pur.wtMM1nlm <- nlsLM( ~ weighted.MM1(rate, conc1, conc.1, Vm, K),
                    data = lisTreat, start = list(Vm = 200, K = 0.1))
print(Pur.wtMM1nlm)
```

```
## Nonlinear regression model
##   model: 0 ~ weighted.MM1(rate, conc1, conc.1, Vm, K)
##    data: lisTreat
##       Vm        K
## 206.8347   0.0546
##  residual sum-of-squares: 14.6
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 1.49e-08
```

```
Pur.wtMM1nlx <- nlxb( ~ weighted.MM1(rate, conc1, conc.1, Vm, K),
                    data = lisTreat, start = list(Vm = 200, K = 0.1), control=list(japprox="jafwd"))
print(Pur.wtMM1nlx)
```

```
## residual sumsquares =  14.597  on  12 observations
##     after  8    Jacobian and  8 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## Vm               206.835         9.225      22.42  7.003e-10  -5.246e-10       230.7
## K              0.0546112      0.007979      6.845  4.488e-05  -6.713e-06       0.131
```

```
# yet another approach
## Chambers and Hastie (1992) Statistical Models in S  (p. 537):
## If the value of the right side [of formula] has an attribute called
## 'gradient' this should be a matrix with the number of rows equal
## to the length of the response and one column for each parameter.
weighted.MM.grad <- function(resp, conc1, conc.1, Vm, K) {
  conc <- c(conc1, conc.1)
  K.conc <- K+conc
  dy.dV <- conc/K.conc
  dy.dK <- -Vm*dy.dV/K.conc
  pred <- Vm*dy.dV
```

```
  pred.5 <- sqrt(pred)
  dev <- (resp - pred) / pred.5
  Ddev <- -0.5*(resp+pred)/(pred.5*pred)
  attr(dev, "gradient") <- Ddev * cbind(Vm = dy.dV, K = dy.dK)
  dev
}
Pur.wtMM.gradnls <- nls( ~ weighted.MM.grad(rate, conc1, conc.1, Vm, K),
                    data = lisTreat, start = list(Vm = 200, K = 0.1))
print(Pur.wtMM.gradnls)
```

```
## Nonlinear regression model
##   model: 0 ~ weighted.MM.grad(rate, conc1, conc.1, Vm, K)
##    data: lisTreat
##       Vm        K
## 206.8347   0.0546
##  residual sum-of-squares: 14.6
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 3.86e-06
```

```
Pur.wtMM.gradnlm <- nlsLM( ~ weighted.MM.grad(rate, conc1, conc.1, Vm, K),
                        data = lisTreat, start = list(Vm = 200, K = 0.1))
print(Pur.wtMM.gradnlm)
```

```
## Nonlinear regression model
##   model: 0 ~ weighted.MM.grad(rate, conc1, conc.1, Vm, K)
##    data: lisTreat
##       Vm        K
## 206.8347   0.0546
##  residual sum-of-squares: 14.6
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 1.49e-08
```

```
Pur.wtMM.gradnlx <- nlxb( ~ weighted.MM.grad(rate, conc1, conc.1, Vm, K),
                        data = lisTreat, start = list(Vm = 200, K = 0.1),
                        control=list(japprox="jafwd"))
print(Pur.wtMM.gradnlx)
```

```
## residual sumsquares =  14.597  on  12 observations
##     after  8    Jacobian and  8 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## Vm               206.835         9.225      22.42  7.003e-10   -3.99e-10       230.7
## K              0.0546112      0.007979      6.845  4.488e-05  -5.206e-06       0.131
```

```
#3 To display the coefficients for comparison:
## rbind(coef(Pur.wtMMnls), coef(Pur.wtMM1nls), coef(Pur.wtMM.gradnls))
## In this example, there seems no advantage to providing the gradient.
## In other cases, there might be.
```

4. Relative offset and other convergence criteria

Here we will mostly look at the relative offset convergence criterion (ROCC) that was proposed by Bates, Douglas M. and Watts, Donald G. (1981). The primary merit of this test is that it it compares the estimated progress towards a minimum sum of squares with the current value. When this is very small, the method terminates. Both `nls()` and `nlsr` try to use this criterion for terminating the process of minimizing the

objective, though there are minor differences of detail. `nlsr` also includes the option of turning off the ROCC with the control parameter `rofftest=FALSE`. `nlsr::nlfb()` also has a small sum of squares test: if the sum of squares is less than the fourth power of the machine precision, `.Machine$double.eps`, the iteration will terminate unless control `smallsstest=FALSE`. In some extreme problems this may be necessary, and they would likely run until limits on maximum number of evaluations of the sum of squares (control `femax`) or Jacobian (control `jemax`) are exceeded. In other words, we are dealing with problems at the edge of computability.

There is a weakness in the ROCC, in that it does not work well with problems that have a zero sum of squares as the solution, so called zero-residual problems. In October 2020, I suggested a patch to the `stats::nls()` function to avoid a zero-divide in the ROCC of `nls()`, using ideas, but not the exact implementation, already in `nlsr::nlfb()`. Previously, the documentation of `nls()` said:

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.

Overview of the ROCC test

The ROCC test uses the calculated **reduction** in the sum of squared residuals in the linearized sub-problem for the latest iteration and compares this to the current sum of squared residuals. The comparison is made by division of the reduction in the sum of squares to the current sum of squares. Clearly if the denominator is zero, we have the awkwardness of zero divided by zero. This is overcome if we add a positive value `scaleOffset` to the denominator. A default value of 0.0 for `scaleOffset` preserves the legacy behaviour of `nls()`.

While I have used ROCC as if there is a single definition, `nls()` and `nlsr::nffb()` use somewhat different formulas and scalings. However, I believe they result in essentially similar outcomes once the `scaleOffset` safeguard is applied.

Some implementation ideas for the ROCC

Below in Section 5 we will see that the essential iteration in either a Gauss-Newton or Marquardt method finds a least squares solution to the linearized problem

$$A\delta = b$$

where A is either the Jacobian or a modification thereof for a Marquardt statbilization, and b the appropriate negative of the residuals, possibly augmented with zeros according to the needs of the stabilization.

Clearly we can compute the current sum of squared as the cross product $b^T b$, since the zeros do not alter this value. It turns out the QR decomposition gives us a quite convenient computation for the reduction in this value by the current linearized sub-problem. This happens to be the sum of squares of the elements of the vector

$$r_{proj} = Q'b$$

where Q' is the transpose of the orthogonal decomposition matrix Q from the decomposition.

Let us illustrate this with a linear least squares problem. Note that this problem uses an explicit column of 1s. Most "regression" calculations assume a constant term which is included automatically. Furthermore, a linear model with just a single constant minimizes the sum of squares when that constant is the mean of the

dependent variable, that is the mean of column b. This sum of squares is generally called the "Total Sum of Squares" and any linear model with more than a constant will have a smaller some of squares. Here, and in most nonlinear models, we do not have that guarantee. Hence we will talk of the "overall" sum of squares for want of a better term, and this is simply the sum of squares of the elements of b. We will judge our progress by this number.

First we create some data for the A and b.

```
# QRsolveEx.R -- example of solving least squares with QR
# J C Nash 2020-12-06
# Enter matrix from Compact Numerical Methods for Computers, Ex04
text <- c(563, 262, 461, 221, 1, 305,
658, 291, 473, 222, 1, 342,
676, 294, 513, 221, 1, 331,
749, 302, 516, 218, 1, 339,
834, 320, 540, 217, 1, 354,
973, 350, 596, 218, 1, 369,
1079, 386, 650, 218, 1, 378,
1151, 401, 676, 225, 1, 368,
1324, 446, 769, 228, 1, 405,
1499, 492, 870, 230, 1, 438,
1690, 510, 907, 237, 1, 438,
1735, 534, 932, 235, 1, 451,
1778, 559, 956, 236, 1, 485)
mm<-matrix(text, ncol=6, byrow=TRUE) # byrow is critical!
A <- mm[,1:5] # select the matrix
A # display
```

```
##       [,1] [,2] [,3] [,4] [,5]
##  [1,]  563  262  461  221    1
##  [2,]  658  291  473  222    1
##  [3,]  676  294  513  221    1
##  [4,]  749  302  516  218    1
##  [5,]  834  320  540  217    1
##  [6,]  973  350  596  218    1
##  [7,] 1079  386  650  218    1
##  [8,] 1151  401  676  225    1
##  [9,] 1324  446  769  228    1
## [10,] 1499  492  870  230    1
## [11,] 1690  510  907  237    1
## [12,] 1735  534  932  235    1
## [13,] 1778  559  956  236    1
```

```
b <- mm[,6] # rhs
b
```

```
##  [1] 305 342 331 339 354 369 378 368 405 438 438 451 485
```

```
oss<-as.numeric(crossprod(b))
cat("Overall sumsquares of b =",oss,"\n")
```

```
## Overall sumsquares of b = 1960595
```

```
cat("(n-1)*variance=",(length(b)-1)*var(b),"= tss = ",as.numeric(crossprod(b-mean(b))),"\n")
```

```
## (n-1)*variance= 35210 = tss =  35210
```

```
Aqr<-qr(A) # compute the QR decomposition of A
QAqr<-qr.Q(Aqr) # extract the Q matrix of this decomposition
RAqr<-qr.R(Aqr) # This is how to get the R matrix
XAqr<-qr.X(Aqr) # And this is the reconstruction of A from the decomposition
cat("max reconstruction error = ", max(abs(XAqr-A)),"\n")
```

```
## max reconstruction error =  9.0949e-13
```

```
cat("check the orthogonality Q' * Q: ", max(abs(t(QAqr)%*%QAqr-diag(rep(1,dim(A)[2])))),"\n")
```

```
## check the orthogonality Q' * Q:  2.2204e-16
```

```
cat("note failure of orthogonality of Q * Q': ",max(abs(QAqr%*%t(QAqr)-diag(rep(1,dim(A)[1])))),"\n")
```

```
## note failure of orthogonality of Q * Q':  0.874
```

```
Absol<-qr.solve(A,b, tol=1000*.Machine$double.eps) # solve the linear ls problem
Absol # and display it
```

```
## [1]  -0.046192   1.019387  -0.159823  -0.290376 207.782626
```

```
Abres<-qr.resid(Aqr,b)# and get the residual
rss<-as.numeric(crossprod(Abres))
cat("residual sumsquares=",rss,"\n")
```

```
## residual sumsquares= 965.25
```

```
Qtb<-as.vector(t(QAqr)%*%b) # Q' * b -- turns out to be reduction in sumsquares
ssQtb<-as.numeric(crossprod(Qtb))
cat("oss-ssQtb = overall sumsquares minus sumsquares Q' * b = ",oss-ssQtb,"\n")
```

```
## oss-ssQtb = overall sumsquares minus sumsquares Q' * b =  965.25
```

```
cat("Thus reduction in sumsquares is ", ssQtb,"\n")
```

```
## Thus reduction in sumsquares is  1959630
```

```
Qtr<-as.vector(t(QAqr)%*%Abres) # projection of residuals onto range space of A
Qtr
```

```
## [1]  8.8818e-16 -4.4409e-16 -8.8818e-16  8.8818e-16 -4.4409e-16
```

We thus have a quite convenient and available method to compute a relative offset test criterion if we solve our Gauss-Newton or Marquardt sub-problem with a QR decomposition.

Other convergence and termination tests

In order to catch runaway computations where some numerical imprecision causes convergence tests to fail, methods generally check the number of Jacobian or sum of squares (i.e., residual) calculations. `nlsr` does this with the `femax` and `jemax` elements of the `control` list, for which the defaults are 10000 and 5000 respectively, which is possibly overly loose. `minpack.lm::nlsLM` uses control list elements `maxfev` and `maxiter` (default 50) which I believe parallel `femax` and `jemax`. The default value of `maxfev` is 100 times the (number of parameters to estimate + 1). `nls()` appears to have only control element `maxiter` for which the default is 50 also.

While `nls()` and `nlsr` use the ROCC, `minpack.lm` uses a combination of tests:

- control element `ftol` is used in a test that seems to be related to the ROCC, since termination occurs when both the actual and predicted relative reductions in the sum of squares are at most `ftol`.
- when the relative change between two consecutive iterates is at most `ptol` the process terminates.

- if the cosine of the angle between result of evaluation of the residual and any column of the Jacobian is at most `gtol` in absolute value the process terminates. Therefore, `gtol` measures the orthogonality desired between the residual vector and the columns of the Jacobian.

5. Implementation of nonlinear least squares methods

This section is a review of approaches to solving the nonlinear least squares problem that underlies nonlinear regression modelling. In particular, we look at using a QR decomposition for the Levenberg-Marquardt stabilization of the solution of the Gauss-Newton equations.

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

$$f(x) = \sum_i ({r_i}^2)$$

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

$$H(x) = 2(J'J + \sum_i (r_i * Z_i))$$

where J is the Jacobian (first derivatives of r w.r.t. x) and Z_i is the tensor of second derivatives of r_i 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. 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. Choices suggested by Marquardt were $D = I$ (a unit matrix) or $D =$ (diagonal part of $J'J$). The former approach, when λ is large enough that the iteration is essentially

$$\delta = -g/\lambda$$

gives 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-Marquardt_algorithm; Levenberg (1944) predated Marquardt, but the latter seems to have done the practical work that brought the approach to general attention.)

Choices

Both `nlsr::nlxb()` and `minpack.lm::nlsLM` use a Levenberg-Marquardt stabilization of the iteration described above, with `nlsr` using the modification involving the ϕ control parameter. The complexities of the code in `minpack.lm` are such that I have relied largely on the documentation to judge how the iteration is accomplished. `nls()` uses a straightforward Hartley variant of the Gauss-Newton iteration, but rather than form the sum of squares and cross-products, uses a QR decomposition of the matrix J that has been found by a forward difference approximation. The line search used by `nls()` is a simple back-tracking search using a step reduction factor of 0.5 as the default stepsize reduction.

J. C. Nash (1979) and John C. Nash and Walker-Smith (1987) solve

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

by the Cholesky decomposition. In this $D_x = (D + \phi * I)$ as described above and λ is a number of modest size initially. Clearly for $\lambda = 0$ we have a Gauss-Newton method. Typically, the sum of squares of the residuals calculated at the "new" set of parameters is used as a criterion for keeping those parameter values. If so, the size of λ is reduced. If not, we increase the size of λ and compute a new δ. Note that a new J, the expensive step in each iteration, is NOT required in this latter case.

In 2022, a modification to use $D_y = (\psi * D + \phi * I)$ was introduced, though the matrix equations are solved via a QR decomposition approach. Within the code, control parameters `psi`, `phi` and `stepredn` were introduced so that a variety of Gauss-Newton, Hartley, or Marquardt approaches are available by simple control modifications. Experience so far suggests that a Levenberg-Marquardt stabilization is much more reliable than the Gauss-Newton-Hartley choices, but that different selections of `psi` and `phi` perform rather similarly. As for Gauss-Newton methods, the details of how to start, adjust and terminate the iteration lead to many variants, increased by these different possibilities for specifying D. See J. C. Nash (1979).

Using matrix decompositions

In J. C. Nash (1979), 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. Note that we also need to add a zero to the residual vector (the right-hand side) for each diagonal element.

Various authors (including the present one) have suggested different strategies for modification of λ. In the present approach, we reduce the *lambda* parameter before solution. The initial value is the control element `lamda` (note the mis-spelling), which has default 1e-4. If the resulting sum of squares is not reduced, *lambda* is increased, otherwise we move to the next iteration. My current opinion is that a "quick" increase, say a factor of 10, and a "slow" decrease, say a factor of 0.4, work quite well. However, it is worth checking that `lamda` has not got too small or underflowed before applying the increase factor. On the other hand, it is useful to be able to set lamda = 0 along with zero increase and decrease factors `laminc` and `lamdec`

so a Hartley method can be evaluated with the program(s) by setting the control `stepredn`. Regularly, the current code `nlfb()` uses the line

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

to ensure we get an increase. However, these possibilities are really for those of us playing to improve algorithms and not for practitioner use.

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()`.

We could implement the methods using the equations above. However, the accumulation of inner products in J^TJ 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

$$QR = J$$

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

$$R\delta = Q^Tr$$

for which the solution is also the solution of the Gauss-Newton equations. 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 λ 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 identity of order n), but Marquardt (1963) showed that using the diagonal elements of J^TJ for D results in a useful implicit scaling of the parameters. John C. 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^TJ before multiplying by λ 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 I_n to the matrix already augmented matrix. We must also append a further n zeros to the augmented r.

6. Fixed parameters

Motivation for fixed parameters

In finding optimal parameters in nonlinear optimization and nonlinear least squares problems, we may wish to fix one or more parameters while allowing the rest to be adjusted to explore or optimize an objective function. A lot of the material is drawn from Nash J C (2014) **Nonlinear parameter optimization using R tools** Chichester UK: Wiley, in particular chapters 11 and 12.

Background for fixed parameters

Here are some of the ways fixed parameters may be specified in R packages.

- `nlxb()` in package `nlsr` (formerly part of defunct package `nlmrt`) uses a character vector `masked` of quoted parameter names. These parameters will NOT be altered by the algorithm. This approach has a simplicity that is attractive, but requires an extra argument to calling sequences.
- `nlfb()` in `nlsr` (formerly in `nlmrt` also) uses the vector `maskidx` of (integer) 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 nlxb which uses the names of the parameters.

- `Rvmmin` and `Rcgmin` use an indicator vector `bdmsk` that has 1 for each parameter that is "free" or unconstrained, and 0 for any parameter that is fixed or MASKED for the duration of the optimization.

Note that the function `bmchk()` in package `optimx` contains a much more extensive examination of the bounds on parameters. In particular, it considers such issues as:

- inadmissible bounds (lower > upper),
- when to convert a pair of bounds where `upper["parameter"] - lower["parameter"] < tol` to a fixed or masked parameter (`maskadded`), and
- whether parameters outside of bounds should be moved to the nearest bound (`parchanged`).

It may be useful to use **inadmissible** to refer to situations where a lower bound is higher than an upper bound and **infeasible** where a parameter is outside the bounds.

From `optimx` the function `optimr()` can call many different "optimizers" (actually function minimization methods that may include bounds and possibly masks). Masks could be specified by setting the lower and upper bounds equal for the parameters to be fixed. While this may seem to be a simple method for specifying masks, there can be computational as well as conceptual difficulties. For example, what happens when the upper bound is only very slightly greater than the lower bound. Also should we stop or declare an error if starting values are NOT on the fixed value.

Of these choices, my current preference is to use the last one – setting lower and upper bounds equal for fixed parameters, and furthermore setting the starting value to this fixed value, and otherwise declaring an error. The approach does not add any special argument for masking, and is relatively obvious to novice users. However, such users may be tempted to put in narrow bounds rather than explicit equalities, and this could have deleterious consequences.

Internal structures to specify fixed parameters

`bdmsk` is the internal structure used in `Rcgmin` and `Rvmmin` to handle bounds constraints as well as masks. There is one element of `bdmsk` for each parameter, and in `Rcgmin` and `Rvmmin`, this is used on input to specify parameter i as fixed or masked by setting `bdmsk[i] <- 0`. Free parameters have their `bdmsk` element 1, but during optimization in the presence of bounds, we can set other values. The full set is as follows

- 1 for a free or unconstrained parameter
- 0 for a masked or fixed parameter
- -0.5 for a parameter that is out of bounds high (> upper bound)
- -1 for a parameter at its upper bound
- -3 for a parameter at its lower bound
- -3.5 for a parameter that is out of bounds low (< lower bound)

Not all these possibilities will be used by all methods that use `bdmsk`.

The -1 and -3 are historical, and arose in the development of BASIC codes for John C. Nash and Walker-Smith (1987) which is now available for free download at https://archive.org/details/NLPE87plus. In particular, adding 2 gives 1 for an upper bound and -1 for a lower bound, simplifying the expression to decide if an optimization trial step will move away from a bound.

Proposed approaches to fix parameters

Because masks (fixed parameters) reduce the dimension of the optimization problem, we can consider modifying the problem to the lower dimension space. This is Duncan Murdoch's suggestion, using

- `fn0(par0)` to be the initial user function of the full dimension parameter vector `par0`
- `fn1(par1)` to be the reduced or internal functin of the reduced dimension vector `par1`
- `par1 <- forward(par0)`
- `par0 <- inverse(par1)`

The major advantage of this approach is explicit dimension reduction. The main disadvantage is the effort of transformation at every step of an optimization.

An alternative is to use the `bdmsk` vector to **mask** the optimization search or adjustment vector, including gradients and (approximate) Hessian matrices. A 0 element of `bdmsk` "multiplies" any adjustment. The principal difficulty is to ensure we do not essentially divide by zero in applying any inverse Hessian. This approach avoids `forward`, `inverse` and `fn1`. However, it may hide the reduction in dimension, and caution is necessary in using the function and its derived gradient, Hessian and derived information.

Examples of use of fixed parameters

Using `Rvmmin`, we easily minimize the function

$$sq(x) = \sum_i (i - x_i)^2$$

a simple quadratic. The unconstrained solution has the function zero when the parameters are the sequence 1 to the number of parameters. When we fix the first parameter at 0.3, the smallest we can make the function is 0.49, i.e., $(1 - 0.3)^2$. In both cases the `convergence` indicator, 2, means that the gradient at the suggested solution was very small.

```
require(optimx)
sq<-function(x){
   nn<-length(x)
   yy<-1:nn
   f<-sum((yy-x)^2)
#   cat("Fv=",f," at ")
#   print(x)
   f
}
sq.g <- function(x){
   nn<-length(x)
   yy<-1:nn
   gg<- 2*(x - yy)
}
xx <- c(.3, 4)
uncans <- Rvmmin(xx, sq, sq.g)
uncans
```

```
## $par
## [1] 1 2
##
## $value
## [1] 0
##
## $counts
## function gradient
##        4        3
##
## $convergence
## [1] 2
##
## $message
## [1] "Rvmminu appears to have converged"
```

```
mybm <- c(0,1) # fix parameter 1
cans <- Rvmmin(xx, sq, sq.g, bdmsk=mybm)
cans
```

```
## $par
## [1] 0.3 2.0
##
## $value
## [1] 0.49
##
## $counts
## function gradient
##        6        4
##
## $convergence
## [1] 2
##
## $message
## [1] "Rvmminb appears to have converged"
##
## $bdmsk
## [1] 0 1
```

Using the same example for `nlsr::nlfb()`, we get the following.

```
## rm(list=ls()) # clear workspace??
library(nlsr)
sqres<-function(x){
   nn<-length(x)
   yy<-1:nn
   res <- (yy-x)
}
sqjac <- function(x){
   nn<-length(x)
   yy<-1:nn
   JJ <- matrix(-1, nrow=nn, ncol=nn)
   attr(JJ, "gradient") <- JJ ## !! Critical statement
   JJ
}
xx <- c(.3, 4) # repeat for completeness
anlfbu<-nlfb(xx, sqres, sqjac)
anlfbu
```

```
## residual sumsquares =  3.645  on  2 observations
##     after  23    Jacobian and  30 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## p1                -0.35           NA          NA       NaN     3.1e-07           2
## p2                 3.35           NA          NA       NaN     3.1e-07           0
```

```
anlfbc<-nlfb(xx, sqres, sqjac, lower=c(xx[1], -Inf), upper=c(xx[1], Inf), trace=FALSE)
anlfbc
```

```
## residual sumsquares =  0.98  on  2 observations
##     after  24    Jacobian and  24 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## p1                   0.3U M         NA          NA         NA           0          NA
```

```
## p2                   2.7           1.4      1.929     0.3045    1.55e-07       1.414
```

```
# Following will warn All parameters are masked
anlfbe<-try(nlfb(xx, sqres, sqjac, lower=c(xx[1], xx[2]), upper=c(xx[1],xx[2])))
```

```
## Warning in nlfb(xx, sqres, sqjac, lower = c(xx[1], xx[2]), upper = c(xx[1], :
## All parameters are masked
```

It is difficult to run this example with `nlsr::nlxb()`, as we need to provide a formula that spans the "data". Note that the unconstrained problem gives a warning when we try to compute a p-value for an essentially perfect minimum of the sum of squares.

?? Note that Dec 28 version does NOT work, but Dec 03 version does. Issue with changes between 03 and 28, probably in how "formula" is treated. – see stats::model.frame – need to have a version that handles "masked"

```
nn<-length(xx) # Also length(yy)
if (nn != 2) stop("This example has nn=2 only!")
yy<-1:2
v1 <- c(1,0); v2 <- c(0,1)
anlxbu <- nlxb(yy~v1*p1+v2*p2, start=c(p1=0.3, p2=4) )
anlxbu
```

```
## residual sumsquares =  7.182e-17  on  2 observations
##     after  3    Jacobian and  3 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## p1                    1           Inf           0       NaN    -2.8e-09           1
## p2                    2           Inf           0       NaN   7.999e-09           1
```

```
anlxbc <- nlxb(yy~v1*p1+v2*p2, start=c(p1=0.3, p2=4), masked=c("p1") )
anlxbc
```

```
## residual sumsquares =  7.182e-17  on  2 observations
##     after  3    Jacobian and  3 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## p1                    1           Inf           0       NaN    -2.8e-09           1
## p2                    2           Inf           0       NaN   7.999e-09           1
```

We can also use a different example that better illustrates `nlsr::nlxb()`.

```
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)
weedux <- nlxb(weed~b1/(1+b2*exp(-b3*ii)), start=c(b1=200, b2=50, b3=0.3))
weedux
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  6    Jacobian and  6 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1               196.186         11.31       17.35  3.167e-08  -5.069e-11        1011
## b2               49.0916         1.688       29.08  3.284e-10  -2.192e-09      0.4605
## b3               0.31357      0.006863       45.69  5.768e-12    2.61e-07     0.04714
```

```
#- Old mechanism of 'nlsr' NO longer works
#- weedcx <- nlxb(weed~b1/(1+b2*exp(-b3*ii)), start=c(b1=200, b2=50, b3=0.3), masked=c("b1"))
weedcx <- nlxb(weed~b1/(1+b2*exp(-b3*ii)), start=c(b1=200, b2=50, b3=0.3),
               lower=c(b1=200, b2=0, b3=0), upper=c(b1=200, b2=100, b3=40))
weedcx
```

```
## residual sumsquares =  2.6182  on  12 observations
##     after  4    Jacobian and  4 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1                   200U M         NA         NA         NA           0          NA
## b2                49.5108          1.12      44.21  8.421e-13  -2.887e-07        1022
## b3               0.311461      0.002278      136.8  1.073e-17   0.0001635      0.4569
```

```
rfn <- function(bvec, weed=weed, ii=ii){
  res <- rep(NA, length(ii))
  for (i in ii){
    res[i]<- bvec[1]/(1+bvec[2]*exp(-bvec[3]*i))-weed[i]
  }
  res
}
weeduf <- nlfb(start=c(200, 50, 0.3),resfn=rfn,weed=weed, ii=ii, control=list(japprox="jacentral"))
weeduf
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  6    Jacobian and  6 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## p1                196.186         11.31       17.35  3.167e-08   1.378e-11        1011
## p2                49.0916         1.688       29.08  3.284e-10  -2.466e-09      0.4605
## p3                0.31357      0.006863       45.69  5.768e-12   5.067e-07     0.04714
```

```
#- maskidx method to specify masks no longer works
#- weedcf <- nlfb(start=c(200, 50, 0.3),resfn=rfn,weed=weed, ii=ii,
#-                maskidx=c(1), control=list(japprox="jacentral"))
weedcf <- nlfb(start=c(200, 50, 0.3),resfn=rfn,weed=weed, ii=ii, lower=c(200, 0, 0),
               upper=c(200, 100, 40), control=list(japprox="jacentral"))
weedcf
```

```
## residual sumsquares =  2.6182  on  12 observations
##     after  4    Jacobian and  4 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## p1                   200U M         NA         NA         NA           0          NA
## p2                49.5108          1.12      44.21  8.421e-13  -2.888e-07        1022
## p3               0.311461      0.002278      136.8  1.073e-17   0.0001634      0.4569
```

7. Capabilities added to `nlsr` in the 2022 version

In the review of nonlinear modelling tools starting in December 2020, several aspects of `nlsr` were modified.

Numerical approximations to Jacobians

While a defining aspect of `nlsr` is the ability to develop analytic Jacobian functions for `nlfb` to use from the `formula` used in the call to `nlxb`, there are many models for which analytic derivatives are either impossible or, more commonly, simply not available through the table of derivatives in the `nlsr` package. Thus it is important to be able to specify an approximation.

This has been documented above in "2. Analytic versus approximate Jacobians", with examples. Note that the `formula` can use R functions that are not in the derivatives table if we specify a Jacobian approximation.

Self start models

`nls()` and `nlsLM()` allow for self-starting models, but they are not explicitly part of `nlsr::nlxb()`. `nls()` does not need a `start` argument if the `formula` contains as its right-hand side (rhs) the name of a selfStart function that is available in the current search path.

Such selfStart functions often also include code to compute the Jacobian, though this does not seem to be a requirement. We have also noted that some of the selfStart models – particularly those in the base-R file `./src/library/stats/R/zzModels.R` – may simply use a crude start and a different algoithm, such as the 'plinear' option. Thus they are not really developing an approximate start, but conducting an alternative solution.

While I contemplated using the "no start argument" approach to selfStart models, the mechanism chosen to use their capabilities is to invoke the `getInitial()` function to set the starting parameter vector. Moreover, by setting control element `japprox` to 'SSJac', we use the Jacobian code available in the selfStart function.

Here is an example using the Michaelis-Menten model in the `stats` package.

```
cat("=== SSmicmen ===\n")
```

```
## === SSmicmen ===
```

```
dat <- PurTrt <- Puromycin[ Puromycin$state == "treated", ]
frm <- rate ~ SSmicmen(conc, Vm, K)
strt<-getInitial(frm, data = dat)
print(strt)
```

```
##         Vm          K
## 212.683707   0.064121
```

```
fm1x <- nlxb(frm, data = dat, start=strt,
             trace=FALSE, control=list(japprox="SSJac"))
fm1x
```

```
## residual sumsquares =  1195.4  on  12 observations
##     after  3    Jacobian and  3 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## Vm               212.684         6.947      30.61  3.241e-11  -1.246e-10        2050
## K              0.0641213      0.008281      7.743  1.565e-05  -0.0008175       1.574
```

We see very quick convergence using the approximation generated by the SSmicmen code.

`nls()`, in the absence of a selfStart model and with no `start` specified, puts all parameters equal to 1. Since there are situations such as the Lanczos multiple exponential model, i.e.,

```
y ~ a * exp(-b*x) + c * exp(-d*x) + e * exp(-f*x)
```

where the three terms would be equivalent with parameters `a` through `f` all at 1, a minor modification to set the parameters so they are all different seems more appropriate. While this could be automated, I prefer to insist that the user take responsibility for an initial guess in these cases.

Models with partially linear parameters

`nls()` offers the choice of solving a problem with partially linear parameters with a version of the Golub-Pereyra VARPRO algorithm. The choice is specified in by `algorithm="plinear"` in the call. (Note that `algorithm="port"` is also possible and replaces the default Gauss-Newton method with the `nl2sol` code from the Bell Port library (Fox (1997), Dennis, Gay, and Welsch (1981)). However, "port" does not provide for partially linear parameters, though it does allow bounds constraints on them.)

Unfortunately, at least in my opinion, when the algorithm is changed to "plinear", the user must also change the formula that specifies the model to be fitted! This is illustrated with one of the examples from the manual page of `nls()`. Note how asking for the `plinear` option introduces another parameter into the model. We can set up the computation with this parameter explicitly included when using the default algorithm, which I believe is a more transparent choice. However, the conditional linearity (VARPRO) algorithm is generally more reliable in getting the optimal parameters in difficult cases. Ideally, the details of use of the "plinear" approach would be hidden from view. Accomplishing that remains an open issue.

```r
## Comment in example is "## using conditional linearity"
DNase1 <- subset(DNase, Run == 1) # data
## using nls with plinear
# Using a formula that explicitly includes the asymptote. (Default algorithm.)
fm2DNase1orig <- nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(Asym = 10, xmid = 0, scal = 1))
summary(fm2DNase1orig)
```

```
##
## Formula: density ~ Asym/(1 + exp((xmid - log(conc))/scal))
##
## Parameters:
##      Estimate Std. Error t value Pr(>|t|)
## Asym   2.3452     0.0782    30.0  2.2e-13 ***
## xmid   1.4831     0.0814    18.2  1.2e-10 ***
## scal   1.0415     0.0323    32.3  8.5e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0192 on 13 degrees of freedom
##
## Number of iterations to convergence: 6
## Achieved convergence tolerance: 2.79e-07
```

```r
# Using conditional linearity. Note the linear paraemter does not appear explicitly.
#  And we must change the starting vector.
fm2DNase1 <- nls(density ~ 1/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(xmid = 0, scal = 1),
                 algorithm = "plinear")
summary(fm2DNase1)
```

```
##
## Formula: density ~ 1/(1 + exp((xmid - log(conc))/scal))
##
## Parameters:
##      Estimate Std. Error t value Pr(>|t|)
## xmid   1.4831     0.0814    18.2  1.2e-10 ***
## scal   1.0415     0.0323    32.3  8.5e-14 ***
## .lin   2.3452     0.0782    30.0  2.2e-13 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0192 on 13 degrees of freedom
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 1.11e-06
```

```r
# How to run with "port" algorithm -- NOT RUN
# fm2DNase1port <- nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
#                  data = DNase1,
#                  start = list(Asym = 10, xmid = 0, scal = 1),
#                  algorithm = "port")
# summary(fm2DNase1port)
```

```
# require(minpack.lm) # Does NOT offer VARPRO. First example is WRONG. NOT RUN.
# fm2DNase1mA <- nlsLM(density ~ 1/(1 + exp((xmid - log(conc))/scal)),
#                   data = DNase1,
#                   start = list(xmid = 0, scal = 1))
# summary(fm2DNase1mA)
# fm2DNase1mB <- nlsLM(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
#                   data = DNase1,
#                   start = list(Asym=10, xmid = 0, scal = 1))
# summary(fm2DNase1mB)
#
# # This gets wrong answer. NOT RUN
# fm2DNase1xA <- nlxb(density ~ 1/(1 + exp((xmid - log(conc))/scal)),
#                   data = DNase1,
#                   start = list(xmid = 0, scal = 1))
# print(fm2DNase1xA)
# Original formula with nlsr::nlxb().
fm2DNase1xB <- nlxb(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(Asym=10, xmid = 0, scal = 1))
print(fm2DNase1xB)
```

```
## residual sumsquares =  0.0047896  on  16 observations
##     after  7    Jacobian and  7 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## Asym             2.34518       0.07815      30.01  2.165e-13  -2.378e-11       2.105
## xmid             1.48309       0.08135      18.23  1.219e-10  -9.163e-09       1.454
## scal             1.04145       0.03227      32.27  8.507e-14   2.325e-08      0.1651
```

There are, however, problems where being able to exploit the partial linearity is an advantage. It would be very nice to be able to use the capability WITHOUT having to adjust the model specification.

8. Capabilities still missing from `nlsr`

This section last updated 2021-02-19.

Automatic differentiation of functional models

This is the natural extension of Multi-line expressions. The authors would welcome collaboration with someone who has expertise in this area. Some progress has been made with the `autodiffr` package of Changcheng Li (see https://github.com/Non-Contradiction/autodiffr).

At the time of writing, functional models require that `nlxb` be called with the control `japprox` set to an available approximating function, as discussed above.

Indexed parameters

There is an example in nls.Rd where indexed parameters are used. That is, parameters can be given a subscript e.g., `b[2]`. As far as I can determine, this facility is not documented, and neither `minpack.lm::nlsLM` nor `nlsr::nlxb` can do this.

Here is the example in the `nls()` documentation.

```
## The muscle dataset in MASS is from an experiment on muscle
## contraction on 21 animals.  The observed variables are Strip
## (identifier of muscle), Conc (Cacl concentration) and Length
## (resulting length of muscle section).
```

```r
utils::data(muscle, package = "MASS")

## The non linear model considered is
##       Length = alpha + beta*exp(-Conc/theta) + error
## where theta is constant but alpha and beta may vary with Strip.

with(muscle, table(Strip)) # 2, 3 or 4 obs per strip
```

```
## Strip
## S01 S02 S03 S04 S05 S06 S07 S08 S09 S10 S11 S12 S13 S14 S15 S16 S17 S18 S19 S20
##   4   4   4   3   3   3   2   2   2   2   3   2   2   2   2   4   4   3   3   3
## S21
##   3
```

```r
## We first use the plinear algorithm to fit an overall model,
## ignoring that alpha and beta might vary with Strip.

musc.1 <- nls(Length ~ cbind(1, exp(-Conc/th)), muscle,
              start = list(th = 1), algorithm = "plinear")
summary(musc.1)
```

```
##
## Formula: Length ~ cbind(1, exp(-Conc/th))
##
## Parameters:
##       Estimate Std. Error t value Pr(>|t|)
## th       0.608      0.115    5.31  1.9e-06 ***
## .lin1   28.963      1.230   23.55  < 2e-16 ***
## .lin2  -34.227      3.793   -9.02  1.4e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.67 on 57 degrees of freedom
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 9.33e-07
```

```r
## Then we use nls' indexing feature for parameters in non-linear
## models to use the conventional algorithm to fit a model in which
## alpha and beta vary with Strip.  The starting values are provided
## by the previously fitted model.
## Note that with indexed parameters, the starting values must be
## given in a list (with names):
b <- coef(musc.1)
musc.2 <- nls(Length ~ a[Strip] + b[Strip]*exp(-Conc/th), muscle,
              start = list(a = rep(b[2], 21), b = rep(b[3], 21), th = b[1]))
## IGNORE_RDIFF_BEGIN
summary(musc.2)
```

```
##
## Formula: Length ~ a[Strip] + b[Strip] * exp(-Conc/th)
##
## Parameters:
##     Estimate Std. Error t value Pr(>|t|)
## a1    23.454      0.796   29.46  5.0e-16 ***
```

```
## a2    28.302      0.793   35.70  < 2e-16 ***
## a3    30.801      1.716   17.95  1.7e-12 ***
## a4    25.921      3.016    8.60  1.4e-07 ***
## a5    23.201      2.891    8.02  3.5e-07 ***
## a6    20.120      2.435    8.26  2.3e-07 ***
## a7    33.595      1.682   19.98  3.0e-13 ***
## a8    39.053      3.753   10.41  8.6e-09 ***
## a9    32.137      3.318    9.69  2.5e-08 ***
## a10   40.005      3.336   11.99  1.0e-09 ***
## a11   36.190      3.109   11.64  1.6e-09 ***
## a12   36.911      1.839   20.07  2.8e-13 ***
## a13   30.635      1.700   18.02  1.6e-12 ***
## a14   34.312      3.495    9.82  2.0e-08 ***
## a15   38.395      3.375   11.38  2.3e-09 ***
## a16   31.226      0.886   35.26  < 2e-16 ***
## a17   31.230      0.821   38.02  < 2e-16 ***
## a18   19.998      1.011   19.78  3.6e-13 ***
## a19   37.095      1.071   34.65  < 2e-16 ***
## a20   32.594      1.121   29.07  6.2e-16 ***
## a21   30.376      1.057   28.74  7.5e-16 ***
## b1   -27.300      6.873   -3.97  0.00099 ***
## b2   -26.270      6.754   -3.89  0.00118 **
## b3   -30.901      2.270  -13.61  1.4e-10 ***
## b4   -32.238      3.810   -8.46  1.7e-07 ***
## b5   -29.941      3.773   -7.94  4.1e-07 ***
## b6   -20.622      3.647   -5.65  2.9e-05 ***
## b7   -19.625      8.085   -2.43  0.02661 *
## b8   -45.780      4.113  -11.13  3.2e-09 ***
## b9   -31.345      6.352   -4.93  0.00013 ***
## b10  -38.599      3.955   -9.76  2.2e-08 ***
## b11  -33.921      3.839   -8.84  9.2e-08 ***
## b12  -38.268      8.992   -4.26  0.00053 ***
## b13  -22.568      8.194   -2.75  0.01355 *
## b14  -36.167      6.358   -5.69  2.7e-05 ***
## b15  -32.952      6.354   -5.19  7.4e-05 ***
## b16  -47.207      9.540   -4.95  0.00012 ***
## b17  -33.875      7.688   -4.41  0.00039 ***
## b18  -15.896      6.222   -2.55  0.02051 *
## b19  -28.969      7.235   -4.00  0.00092 ***
## b20  -36.917      8.033   -4.60  0.00026 ***
## b21  -26.508      7.012   -3.78  0.00149 **
## th     0.797      0.127    6.30  8.0e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.11 on 17 degrees of freedom
##
## Number of iterations to convergence: 8
## Achieved convergence tolerance: 2.19e-06
## IGNORE_RDIFF_END
```

The structure of the call is interesting in that `start` is a **list** and the elements of each part are NOT equal in length, as can bee seen from

```
istart = list(a = rep(b[2], 21), b = rep(b[3], 21), th = b[1])
str(istart)
```

```
## List of 3
##  $ a : Named num [1:21] 29 29 29 29 29 ...
##   ..- attr(*, "names")= chr [1:21] ".lin1" ".lin1" ".lin1" ".lin1" ...
##  $ b : Named num [1:21] -34.2 -34.2 -34.2 -34.2 -34.2 ...
##   ..- attr(*, "names")= chr [1:21] ".lin2" ".lin2" ".lin2" ".lin2" ...
##  $ th: Named num 0.608
##   ..- attr(*, "names")= chr "th"
```

9. Nonlinear equations and other non-modelling problems

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 by methods of the type discussed in this article, in particular codes `nlfb` and `nls.lm`.

Conversely, since our problem is, using the notation already established, equivalent to residuals equal to zero, namely,

$r(x) = 0$

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. However, there are methods for nonlinear equations, some of which (e.g., Hasselman (2013)) are available in R packages, and they may be more appropriate. On the other hand, if the nonlinear least squares tools are familiar, it may be more human-efficient to use them, at least as a first try.

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.

```
# NOTE: This is OLD material and not consistent in usage with rest of vignette
library(knitr)
# try different ways of supplying data to R nls stuff
ydata <- 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)
ttdata <- seq_along(ydata) # for testing
mydata <- data.frame(y = ydata, tt = ttdata)
hobsc <- y ~ 100*b1/(1 + 10*b2 * exp(-0.1 * b3 * tt))
ste <- c(b1 = 2, b2 = 5, b3 = 3)

# let's try finding the variables
findmainenv <- function(formula, prm) {
  vn <- all.vars(formula)
  pnames <- names(prm)
  ppos <- match(pnames, vn)
  datvar <- vn[-ppos]
  cat("Data variables:")
  print(datvar)
  cat("Are the variables present in the current working environment?\n")
  for (i in seq_along(datvar)){
    cat(datvar[[i]]," : present=",exists(datvar[[i]]),"\n")
  }
}

findmainenv(hobsc, ste)
```

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

```
y <- ydata
tt <- ttdata
findmainenv(hobsc, ste)
```

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

```
rm(y)
rm(tt)

# ================================


# let's try finding the variables in dotargs
finddotargs <- function(formula, prm, ...) {
  dots <- list(...)
  cat("dots:")
  print(dots)
```

```
  cat("names in dots:")
  dtn <- names(dots)
  print(dtn)
  vn <- all.vars(formula)
  pnames <- names(prm)
  ppos <- match(pnames, vn)
  datvar <- vn[-ppos]
  cat("Data variables:")
  print(datvar)
  cat("Are the variables present in the dot args?\n")
  for (i in seq_along(datvar)){
    dname <- datvar[[i]]
    cat(dname," : present=",(dname %in% dtn),"\n")
  }
}

finddotargs(hobsc, ste, y=ydata, tt=ttdata)
```

```
## 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
```

```
# ================================

y <- ydata
tt <- ttdata
tryq <- try(nlsquiet <- nls(formula=hobsc, start=ste))
if (class(tryq) != "try-error") {print(nlsquiet)} else {cat("try-error\n")}
```

```
## 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
```

```
#- OK
rm(y); rm(tt)

## this will fail
tdots1<-try(nlsdots <- nls(formula=hobsc, start=ste, y=ydata, tt=ttdata))
```

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

```
##   parameters without starting value in 'data': y, tt
```

```
# But ...
y <- ydata # put data in globalenv
tt <- ttdata
tdots2<-try(nlsdots <- nls(formula=hobsc, start=ste))
tdots2
```

```
## 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
```

```
rm(y); rm(tt)
## but ...
mydata<-data.frame(y=ydata, tt=ttdata)
tframe <- try(nlsframe <- nls(formula=hobsc, start=ste, data=mydata))
if (class(tframe) != "try-error") {print(nlsframe)} else {cat("try-error\n")}
```

```
## 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
```

```
#- OK

y <- ydata
tt <- ttdata
tquiet <- try(nlsrquiet <- nlxb(formula=hobsc, start=ste))
if ( class(tquiet) != "try-error") {print(nlsrquiet)}
```

```
## residual sumsquares =  2.5873  on  12 observations
##     after  6    Jacobian and  6 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1              1.96186        0.1131       17.35  3.167e-08  -1.073e-11       130.1
## b2              4.90916        0.1688       29.08  3.284e-10  -7.696e-09       6.165
## b3               3.1357       0.06863       45.69  5.768e-12    1.17e-08       2.735
```

```
#- OK
rm(y); rm(tt)
test <- try(nlsrdots <- nlxb(formula=hobsc, start=ste, y=ydata, tt=ttdata))
```

```
## Error in eval(residexpr, envir = localdata) : object 'tt' not found
```

```
#- Note -- does NOT work -- do we need to specify the present env. in nlfb for y, tt??
if (class(test) != "try-error") { print(nlsrdots) } else {cat("Try error\n") }
```

```
## Try error
```

```
test2 <- try(nlsframe <- nls(formula=hobsc, start=ste, data=mydata))
if (class(test) != "try-error") {print(nlsframe) } else {cat("Try error\n") }
```

```
## Try error
```

```
#- OK


library(minpack.lm)
y <- ydata
tt <- ttdata
nlsLMquiet <- nlsLM(formula=hobsc, start=ste)
print(nlsLMquiet)
```

```
## 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
```

```
#- OK
rm(y)
rm(tt)
## Dotargs
## Following fails
## tdots <- try(nlsLMdots <- nlsLM(formula=hobsc, start=ste, y=ydata, tt=ttdata))
## but ...
tdots <- try(nlsLMdots <- nlsLM(formula=hobsc, start=ste, data=mydata))
if (class(tdots) != "try-error") { print(nlsLMdots) } else {cat("try-error\n") }
```

```
## 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
```

```
#-  Note -- does NOT work
## dataframe
tframe <- try(nlsLMframe <- nlsLM(formula=hobsc, start=ste, data=mydata) )
if (class(tdots) != "try-error") {print(nlsLMframe)} else {cat("try-error\n") }
```

```
## 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
```

```
#- does not work

## detach("package:nlsr", unload=TRUE)
## Uses nlmrt here for comparison
## library(nlmrt)
## txq <- try( nlxbquiet <- nlxb(formula=hobsc, start=ste))
## if (class(txq) != "try-error") {print(nlxbquiet)} else { cat("try-error\n")}
#- Note -- does NOT work
## txdots <- try( nlxbdots <- nlxb(formula=hobsc, start=ste, y=y, tt=tt) )
## if (class(txdots) != "try-error") {print(nlxbdots)} else {cat("try-error\n")}
#- Note -- does NOT work
## dataframe
## nlxbframe <- nlxb(formula=hobsc, start=ste, data=mydata)
## print(nlxbframe)
#- OK
```

Appendix B: Derivative approximation in nls()

This Appendix could benefit from some examples.

From nls.R

```
numericDeriv <- function(expr, theta, rho = parent.frame(), dir=1.0)
{
    dir <- rep_len(dir, length(theta))
    val <- .Call(C_numeric_deriv, expr, theta, rho, dir)
    valDim <- dim(val)
    if (!is.null(valDim)) {
        if (valDim[length(valDim)] == 1)
            valDim <- valDim[-length(valDim)]
        if(length(valDim) > 1L)
            dim(attr(val, "gradient")) <- c(valDim,
                                            dim(attr(val, "gradient"))[-1L])
    }
    val
}
```

From nls.c

```
/*
 *  call to numeric_deriv from R -
 *  .Call("numeric_deriv", expr, theta, rho)
 *  Returns: ans
 */
SEXP
numeric_deriv(SEXP expr, SEXP theta, SEXP rho, SEXP dir)
{
    SEXP ans, gradient, pars;
    double eps = sqrt(DOUBLE_EPS), *rDir;
    int start, i, j, k, lengthTheta = 0;

    if(!isString(theta))
    error(_("'theta' should be of type character"));
    if (isNull(rho)) {
    error(_("use of NULL environment is defunct"));
    rho = R_BaseEnv;
    } else
    if(!isEnvironment(rho))
        error(_("'rho' should be an environment"));
    PROTECT(dir = coerceVector(dir, REALSXP));
    if(TYPEOF(dir) != REALSXP || LENGTH(dir) != LENGTH(theta))
    error(_("'dir' is not a numeric vector of the correct length"));
    rDir = REAL(dir);

    PROTECT(pars = allocVector(VECSXP, LENGTH(theta)));

    PROTECT(ans = duplicate(eval(expr, rho)));

    if(!isReal(ans)) {
    SEXP temp = coerceVector(ans, REALSXP);
    UNPROTECT(1);
```

```
    PROTECT(ans = temp);
    }
    for(i = 0; i < LENGTH(ans); i++) {
    if (!R_FINITE(REAL(ans)[i]))
        error(_("Missing value or an infinity produced when evaluating the model"));
    }
    const void *vmax = vmaxget();
    for(i = 0; i < LENGTH(theta); i++) {
    const char *name = translateChar(STRING_ELT(theta, i));
    SEXP s_name = install(name);
    SEXP temp = findVar(s_name, rho);
    if(isInteger(temp))
        error(_("variable '%s' is integer, not numeric"), name);
    if(!isReal(temp))
        error(_("variable '%s' is not numeric"), name);
    if (MAYBE_SHARED(temp)) /* We'll be modifying the variable, so need to make sure it's unique PR#158
        defineVar(s_name, temp = duplicate(temp), rho);
    MARK_NOT_MUTABLE(temp);
    SET_VECTOR_ELT(pars, i, temp);
    lengthTheta += LENGTH(VECTOR_ELT(pars, i));
    }
    vmaxset(vmax);
    PROTECT(gradient = allocMatrix(REALSXP, LENGTH(ans), lengthTheta));

    for(i = 0, start = 0; i < LENGTH(theta); i++) {
    for(j = 0; j < LENGTH(VECTOR_ELT(pars, i)); j++, start += LENGTH(ans)) {
        SEXP ans_del;
        double origPar, xx, delta;

        origPar = REAL(VECTOR_ELT(pars, i))[j];
        xx = fabs(origPar);
        delta = (xx == 0) ? eps : xx*eps;
        REAL(VECTOR_ELT(pars, i))[j] += rDir[i] * delta;
        PROTECT(ans_del = eval(expr, rho));
        if(!isReal(ans_del)) ans_del = coerceVector(ans_del, REALSXP);
        UNPROTECT(1);
        for(k = 0; k < LENGTH(ans); k++) {
        if (!R_FINITE(REAL(ans_del)[k]))
            error(_("Missing value or an infinity produced when evaluating the model"));
        REAL(gradient)[start + k] =
            rDir[i] * (REAL(ans_del)[k] - REAL(ans)[k])/delta;
        }
        REAL(VECTOR_ELT(pars, i))[j] = origPar;
    }
    }
    setAttrib(ans, install("gradient"), gradient);
    UNPROTECT(4);
    return ans;
}
```

nlsr::numericDerivR.R

This is a replacement for the nls() function numericDeriv() that is coded all in R.

```
numericDerivR <- function(expr, theta, rho = parent.frame(), dir = 1,
                  eps = .Machine$double.eps ^ (1/if(central) 3 else 2), central = FALSE)
## Note: Must this expr must be set up as a call to work properly?
## We set eps conditional on central. But central set AFTER eps. Is this OK?
{   ## cat("numericDeriv-Alt\n")
    dir <- rep_len(dir, length(theta))
    stopifnot(is.finite(eps), eps > 0)
##    rho1 <- new.env(FALSE, rho, 0)
    if (!is.character(theta) ) {stop("'theta' should be of type character")}
    if (is.null(rho)) {
            stop("use of NULL environment is defunct")
            #        rho <- R_BaseEnv;
    } else {
          if(! is.environment(rho)) {stop("'rho' should be an environment")}
          #    int nprot = 3;
    }
    if( ! ((length(dir) == length(theta) ) & (is.numeric(dir) ) ) )
              {stop("'dir' is not a numeric vector of the correct length") }
    if(is.na(central)) { stop("'central' is NA, but must be TRUE or FALSE") }
    res0 <- eval(expr, rho) # the base residuals. C has a check for REAL ANS=res0
    nt <- length(theta) # number of parameters
    mr <- length(res0) # number of residuals
    JJ <- matrix(NA, nrow=mr, ncol=nt) # Initialize the Jacobian
    for (j in 1:nt){
       origPar<-get(theta[j],rho) # This is parameter by NAME, not index
       xx <- abs(origPar)
       delta <- if (xx == 0.0) {eps} else { xx*eps }
       ## JN: I prefer eps*(xx + eps)  which is simpler?
       prmx<-origPar+delta*dir[j]
       assign(theta[j],prmx,rho)
       res1 <- eval(expr, rho) # new residuals (forward step)
#       cat("res1:"); print(res1)
       if (central) { # compute backward step resids for central diff
          prmb <- origPar - dir[j]*delta
          assign(theta[j], prmb, envir=rho) # may be able to make more efficient later?
          resb <- eval(expr, rho)
          JJ[, j] <- dir[j]*(res1-resb)/(2*delta) # vectorized
       } else { ## forward diff
          JJ[, j] <- dir[j]*(res1-res0)/delta
       }  # end forward diff
       assign(theta[j],origPar, rho) # reset value
    } # end loop over the parameters
    attr(res0, "gradient") <- JJ
    return(res0)
}
```

Let us compute the Jacobian for the Hobbs problem with this function and its `nls()` original. I find the mechanism for these functions awkward.

```
library(nlsr) # so we have numericDerivR code
# Data for Hobbs problem
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) # for testing
tt  <-  seq_along(weed) # for testing
```

```
# A simple starting vector -- must have named parameters for nlxb, nls, wrapnlsr.
st  <-  c(b1=1, b2=1, b3=1)
wmodu  <-   weed ~ b1/(1+b2*exp(-b3*tt))
weeddf  <-  data.frame(weed=weed, tt=tt)
weedenv <- list2env(weeddf)
weedenv$b1 <- st[[1]]
weedenv$b2 <- st[[2]]
weedenv$b3 <- st[[3]]
rexpr<-call("-",wmodu[[3]], wmodu[[2]])
r0<-eval(rexpr, weedenv)
cat("Sumsquares at 1,1,1 is ",sum(r0^2),"\n")## Another way
```

```
## Sumsquares at 1,1,1 is  23521
```

```
expr <- wmodu
rho <- weedenv
rexpr<-call("-",wmodu[[3]], wmodu[[2]])
res0<-eval(rexpr, rho) # the base residuals
res0
```

```
##  [1]  -4.5769  -6.3592  -8.6854 -11.8840 -16.0757 -22.1945 -30.4439 -37.5583
##  [9] -49.1561 -61.9480 -74.9950 -90.9720
```

```
## Try the numericDeriv option
theta<-names(st)
nDnls<-numericDeriv(rexpr, theta, weedenv)
nDnls
```

```
##  [1]  -4.5769  -6.3592  -8.6854 -11.8840 -16.0757 -22.1945 -30.4439 -37.5583
##  [9] -49.1561 -61.9480 -74.9950 -90.9720
## attr(,"gradient")
##           [,1]        [,2]       [,3]
##  [1,] 0.73106 -1.9661e-01 1.9661e-01
##  [2,] 0.88080 -1.0499e-01 2.0999e-01
##  [3,] 0.95257 -4.5177e-02 1.3553e-01
##  [4,] 0.98201 -1.7663e-02 7.0651e-02
##  [5,] 0.99331 -6.6481e-03 3.3240e-02
##  [6,] 0.99753 -2.4664e-03 1.4799e-02
##  [7,] 0.99909 -9.1028e-04 6.3715e-03
##  [8,] 0.99966 -3.3569e-04 2.6817e-03
##  [9,] 0.99988 -1.2350e-04 1.1106e-03
## [10,] 0.99995 -4.5300e-05 4.5395e-04
## [11,] 0.99998 -1.7166e-05 1.8311e-04
## [12,] 0.99999 -5.7220e-06 7.4387e-05
```

```
nDnlsR<-numericDerivR(rexpr, theta, weedenv)
nDnlsR
```

```
##  [1]  -4.5769  -6.3592  -8.6854 -11.8840 -16.0757 -22.1945 -30.4439 -37.5583
##  [9] -49.1561 -61.9480 -74.9950 -90.9720
## attr(,"gradient")
##           [,1]        [,2]       [,3]
##  [1,] 0.73106 -1.9661e-01 1.9661e-01
##  [2,] 0.88080 -1.0499e-01 2.0999e-01
##  [3,] 0.95257 -4.5177e-02 1.3553e-01
##  [4,] 0.98201 -1.7663e-02 7.0651e-02
##  [5,] 0.99331 -6.6481e-03 3.3240e-02
```

```
##  [6,] 0.99753 -2.4664e-03 1.4799e-02
##  [7,] 0.99909 -9.1028e-04 6.3715e-03
##  [8,] 0.99966 -3.3569e-04 2.6817e-03
##  [9,] 0.99988 -1.2350e-04 1.1106e-03
## [10,] 0.99995 -4.5300e-05 4.5395e-04
## [11,] 0.99998 -1.7166e-05 1.8311e-04
## [12,] 0.99999 -5.7220e-06 7.4387e-05
```

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

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

Principal differences

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

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

Derivative information

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

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

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

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

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

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

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

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

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

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

Timing comparisons Let us compare timings on the (scaled) Hobbs weed problem.

```
require(microbenchmark)
## nls on Hobbs scaled model
wmods  <-   weed ~ 100*b1/(1+10*b2*exp(-0.1*b3*tt))
stx<-c(b1=2, b2=5, b3=3)
tnls<-microbenchmark((anls<-nls(wmods, start=stx, data=weeddf)), unit="us")
tnls
```

```
## Unit: microseconds
##                                             expr    min    lq   mean median
##  (anls <- nls(wmods, start = stx, data = weeddf)) 841.92 851.6 866.28 854.54
##     uq    max neval
##  858.9 1254.8   100
```

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

```
## Unit: microseconds
##                                                  expr    min     lq   mean
##  (anlxb <- nlsr::nlxb(wmods, start = stx, data = weeddf)) 1227.6 1244.8 1289.5
##  median     uq    max neval
##  1273.8 1317.8 1607.8   100
```

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

```
## Unit: microseconds
##                                                                      expr
##  (anlsLM <- minpack.lm::nlsLM(start = stx, formula = wmods, data = weeddf))
##     min    lq   mean median     uq    max neval
##  623.75 629.4 640.55 631.95 635.24 1054.4   100
```

Programmatic modelling functions

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

This limitation could be overcome using appropriate automatic differentiation code (to provide derivative computations based on transformation of the modelling function's programmatic form). The present work-around is to use numerical approximation by specifying the control element `japprox`.

Functional expression of residuals and Jacobian

```
require(microbenchmark)
## nlsr::nlfb() on Hobbs scaled
tnlfb<-microbenchmark((anlfb<-nlsr::nlfb(start=st1, resfn=shobbs.res, jacfn=shobbs.jac)), unit="us")
tnlfb
```

```
## Unit: microseconds
##                                                                      expr
##  (anlfb <- nlsr::nlfb(start = st1, resfn = shobbs.res, jacfn = shobbs.jac))
##   min     lq mean median     uq    max neval
##  4252 4349.1 4527 4432.2 4513.4 8143.7   100
```

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

```
## Unit: microseconds
##                                                                          expr
##  (anls.lm <- minpack.lm::nls.lm(par = st1, fn = shobbs.res, jac = shobbs.jac))
##     min     lq   mean median     uq    max neval
##  111.94 113.68 120.53 114.91 115.98 361.02   100
```

Marquardt stabilization

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

`y ~ (some expression)`

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

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

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

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

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

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

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

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

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

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

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

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

Criterion used to terminate the iteration

`nls()` and `nlsr` use a form of the relative offset convergence criterion, Bates, Douglas M. and Watts, Donald G. (1981). `minpack.lm` uses a somewhat different and more complicated set of tests. Unfortunately, the relative offset criterion as implemented in `nls()` is unsuited to problems where the residuals can be zero. As of R 4.1.0, there is a work-around in providing a non-zero value to the control element `scaleOffset` as documented in the manual page of `nls()`. See *An illustrative nonlinear regression problem* below.

Output of the modelling functions

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

```
str(nlsy0t0ax)
```

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

```
##   ..- attr(*, "names")= chr "t0a"
##  $ control    :List of 7
##   ..$ maxiter    : num 10000
##   ..$ tol        : num 1e-05
##   ..$ minFactor  : num 0.000977
##   ..$ printEval  : logi FALSE
##   ..$ warnOnly   : logi FALSE
##   ..$ scaleOffset: num 0
##   ..$ nDcentral  : logi FALSE
##  - attr(*, "class")= chr "nls"
```

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

```
str(nlsry1t0a)
```

```
## List of 13
##  $ resid       : num [1:20] 4.00e-05 -2.03e-09 -1.70e-09 -1.41e-09 -1.16e-09 ...
##   ..- attr(*, "gradient")= num [1:20, 1:2] 0.00001 1 1.18921 1.31607 1.41421 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : NULL
##   .. .. ..$ : chr [1:2] "a" "b"
##  $ jacobian    : num [1:20, 1:2] 0.00001 1 1.18921 1.31607 1.41421 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:2] "a" "b"
##  $ feval       : num 7
##  $ jeval       : num 7
##  $ coefficients: Named num [1:2] 4 0.25
##   ..- attr(*, "names")= chr [1:2] "a" "b"
##  $ ssquares    : num 1.6e-09
##  $ lower       : num [1:2] -Inf -Inf
##  $ upper       : num [1:2] Inf Inf
##  $ maskidx     : int(0)
##  $ weights     : num [1:20] 1 1 1 1 1 1 1 1 1 1 ...
##  $ formula     :Class 'formula'  language y1 ~ a * (t0a^b)
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##  $ resfn       :function (prm)
##  $ data        : symbol edta
##  - attr(*, "class")= chr "nlsr"
```

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

Prediction

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

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

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

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

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

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

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

An illustrative nonlinear regression problem

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

```
# Here we set up an example problem with data
# Define independent variable
t0 <- 0:19
t0a<-t0
t0a[1]<-1e-20 # very small value
# Drop first value in vectors
t0t<-t0[-1]
y1 <- 4 * (t0^0.25)
y1t<-y1[-1]
n <- length(t0)
fuzz <- rnorm(n)
range <- max(y1)-min(y1)
## add some "error" to the dependent variable
y1q <- y1 + 0.2*range*fuzz
edta <- data.frame(t0=t0, t0a=t0a, y1=y1, y1q=y1q)
edtat <- data.frame(t0t=t0t, y1t=y1t)
```

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

"Warning

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

It goes on to recommend that users add "error" to the data to avoid such problems. I feel this is a very unsatisfactory kludge. It is NOT due to a genuine mathematical issue, but due to the relative offset convergence criterion used to terminate the method. Using the contr

Here is the output.

```
cprint <- function(obj){
   # print object if it exists
  sobj<-deparse(substitute(obj))
  if (exists(sobj)) {
      print(obj)
  } else {
      cat(sobj," does not exist\n")
```

```
  }
#  return(NULL)
}
start1 <- c(a=1, b=1)
try(nlsy0t0 <- nls(formula=y1~a*(t0^b), start=start1, data=edta))
```

nls

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

```
cprint(nlsy0t0)
```

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

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

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

```
cprint(nlsy0t0x)
```

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

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

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

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

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

```
cprint(nlsy0t0t)
```

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

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

nlsr

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

```
cprint(nlsry1t0)
```

```
## [1] "Error in model2rjfun(formula, pnum, data = data) : Jacobian contains NaN\n"
## attr(,"class")
```

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

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

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

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

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

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

minpack.lm

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

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

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

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

```
## Nonlinear regression model
##   model: y1t ~ a * (t0t^b)
##    data: edtat
```

```
##    a    b
## 4.00 0.25
##  residual sum-of-squares: 0
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 1.49e-08
```

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

Problems that are NOT regressions

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

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

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

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

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

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

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

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

```
## Nonlinear regression model
##   model: y ~ ((x1 + t * x2 - exp(t))^2 + (x3 + x4 * sin(t) - cos(t))^2)
##    data: bddata
##      x1      x2      x3      x4
## -11.594  13.204  -0.403   0.237
##  residual sum-of-squares: 85822
```

```
##
## Number of iterations to convergence: 867
## Achieved convergence tolerance: 9.76e-06
```

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

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

```
nlsLMbd10k
```

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

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

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

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

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

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

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

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

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

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

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

A check on the Brown and Dennis calculation via function minimization

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

```
#' Brown and Dennis Function
#'
#' Test function 16 from the More', Garbow and Hillstrom paper.
#'
```

```
#' The objective function is the sum of \code{m} functions, each of \code{n}
#' parameters.
#'
#' \itemize{
#'   \item Dimensions: Number of parameters \code{n = 4}, number of summand
#'   functions \code{m >= n}.
#'   \item Minima: \code{f = 85822.2} if \code{m = 20}.
#' }
#'
#' @param m Number of summand functions in the objective function. Should be
#'   equal to or greater than 4.
#' @return A list containing:
#' \itemize{
#'   \item \code{fn} Objective function which calculates the value given input
#'   parameter vector.
#'   \item \code{gr} Gradient function which calculates the gradient vector
#'   given input parameter vector.
#'   \item \code{fg} A function which, given the parameter vector, calculates
#'   both the objective value and gradient, returning a list with members
#'   \code{fn} and \code{gr}, respectively.
#'   \item \code{x0} Standard starting point.
#' }
#' @references
#' More', J. J., Garbow, B. S., & Hillstrom, K. E. (1981).
#' Testing unconstrained optimization software.
#' \emph{ACM Transactions on Mathematical Software (TOMS)}, \emph{7}(1), 17-41.
#' \url{https://doi.org/10.1145/355934.355936}
#'
#' Brown, K. M., & Dennis, J. E. (1971).
#' \emph{New computational algorithms for minimizing a sum of squares of
#' nonlinear functions} (Report No. 71-6).
#' New Haven, CT: Department of Computer Science, Yale University.
#'
#' @examples
#' # Use 10 summand functions
#' fun <- brown_den(m = 10)
#' # Optimize using the standard starting point
#' x0 <- fun$x0
#' res_x0 <- stats::optim(par = x0, fn = fun$fn, gr = fun$gr, method =
#' "L-BFGS-B")
#' # Use your own starting point
#' res <- stats::optim(c(0.1, 0.2, 0.3, 0.4), fun$fn, fun$gr, method =
#' "L-BFGS-B")
#'
#' # Use 20 summand functions
#' fun20 <- brown_den(m = 20)
#' res <- stats::optim(fun20$x0, fun20$fn, fun20$gr, method = "L-BFGS-B")
#' @export
#`
brown_den <- function(m = 20) {
  list(
    fn = function(par) {
      x1 <- par[1]
```

```
    x2 <- par[2]
    x3 <- par[3]
    x4 <- par[4]

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

    ti <- (1:m) * 0.2
    sinti <- sin(ti)
    l <- x1 + ti * x2 - exp(ti)
    r <- x3 + x4 * sinti - cos(ti)
    f <- l * l + r * r
    lf4 <- 4 * l * f
    rf4 <- 4 * r * f
    c(
      sum(lf4),
      sum(lf4 * ti),
      sum(rf4),
      sum(rf4 * sinti)
    )
  },
  fg = function(par) {
    x1 <- par[1]
    x2 <- par[2]
    x3 <- par[3]
    x4 <- par[4]

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

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

    list(
```

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

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

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

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

References

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

Dennis, John E., David M. Gay, and Roy E. Welsch. 1981. "An Adaptive Nonlinear Least-Squares Algorithm." *ACM Transactions on Mathematical Software* 7 (3): 348–68.

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

Fox, Phyllis. 1997. "The Port Mathematical Subroutine Library, Version 3." Murray Hill, NJ: AT&T Bell Laboratories. http://www.bell-labs.com/project/PORT/.

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

Hasselman, Berend. 2013. *Nleqslv: Solve Systems of Non Linear Equations*. http://CRAN.R-project.org/package=nleqslv.

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

Levenberg, Kenneth. 1944. "A Method for the Solution of Certain Non-Linear Problems in Least Squares." *Quarterly of Applied Mathematics* 2: 164--168.

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

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

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

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.

———. 2012. *Nlmrt: Functions for Nonlinear Least Squares Solutions*.

Nash, John C., and Arkajyoti Bhattacharjee. 2022. "Refactoring the 'nls()' function in R." https://github.com/nashjc/RNonlinearLS/blob/main/RefactoringNLS/RefactoringNLS.pdf.

Nash, John C., and Mary Walker-Smith. 1987. *Nonlinear Parameter Estimation: An Integrated System in BASIC*. Book. New York: Marcel Dekker.