Refactoring the `nls()` function in R

John C. Nash [*] Arkajyoti Bhattacharjee [†]

2021-8-20

Contents

Abstract

This article considers the features and limitations of the R function `nls()`. It arose from a Google Summer of Code project "Improvements to `nls()`", a tool intended for the estimation of models written as a formula that has at least one parameter that is "nonlinear". That is, the model is not estimable via solving a finite

[*]retired professor, Telfer School of Management, University of Ottawa
[†]Department of Mathematics and Statistics, Indian Institute of Technology, Kanpur

set of linear equations. A companion document **Variety in Nonlinear Least Squares Codes** presents an overview of methods for the problem which takes a much wider view of the problem of minimizing a function that can be written as a sum of squared terms, and of the even more general case of nonlinear regression.

An important overall consideration in our work has been the maintainability of the code base that supports the `nls()` functionality, as we believe that the existing code makes maintenance and improvement very difficult. Moreover, legacy applications and examples mean some improvments are blocked. Thus we have not addressed all the issues raised, but we believe that the discussion and trials so far are relevant to improving both `nls()` and R generally.

The `nls()` function: strengths and shortcomings

`nls()` is the tool in base R (the distributed software package from CRAN for estimating nonlinear statistical models. The function was developed mainly in the 1980s and 1990s by Doug Bates et al., initially for S (see https://en.wikipedia.org/wiki/S_%28programming_language%29) . The ideas spring primarily from the book by Bates and Watts (1988).

The `nls()` function has a remarkable and quite comprehensive set of capabilities for estimating nonlinear models that are expressed as formulas. In particular, we note that it

- handles formulas that include R functions
- allows data to be subset
- permits parameters to be indexed over a set of related data
- produces measures of variability (i.e., standard error estimates) for the estimated parameters
- has related profiling capabilities for exploring the likelihood surface as parameters are changed

With such a range of features and a long history, it is not surprising that the code has become untidy and overly patched. It is, to our mind, essentially unmaintainable. Moreover, its underlying methods can and should be improved. Let us review some of the issues. We will then propose corrective actions, some of which we have carried out either in demonstrations or in the distributed coed.

Feature: Convergence and termination tests (FIXED)

A previous issue with `nls()` that prevented it from providing parameter estimates for zero-residual (i.e., perfect fit) data was corrected thanks to suggestions by one of us.

In the manual page for `nls()` in R 4.0.0 there is the warning

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

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. This performs well on data of the form

$$y = f(x, \theta) + eps$$

(with `var(eps) > 0`). It fails to indicate convergence on data of the form

$$y = f(x, \theta)$$

because the criterion amounts to comparing two components of the round-off error.

If you wish to test `nls` on artificial data please add a noise component, as shown in the example below.

This amounts to admitting R cannot solve perfectly well-posed problems. The suggestion that one needs to pollute data with errors is one that should be offensive to good science.

It turns out that this issue can be easily resolved. The key "convergence test" – more properly a "termination test" for the **program** rather than for convergence of the underlying **algorithm** – is the Relative Offset

Convergence Criterion (see Bates, Douglas M. and Watts, Donald G. (1981)). This works by projecting the proposed step in the parameter vector on the gradient and estimating how much the sum of squares loss function will decrease. This is divide by the current size of the loss function to avoid scale issues. When we have "converged", the estimated decrease is very small, as usually is its ratio to the sum of squares. However, with small residuals, the sum of squares loss function is (almost) zero and we get the possibility of a zero-divide failure.

Adding a small quantity to the loss function before dividing avoids trouble. In 2021, one of us (J. Nash) proposed that `nls.control()` have an additional parameter `scaleOffset` with a default value of zero. Setting it to a small number – 1.0 is a reasonable choice – allows small-residual problems (i.e., near-exact fits) to be dealt with easily. We call this the **safeguarded relative offset convergence criterion**. The default value gives the legacy behaviour. We are pleased to note that this improvement is now in the R distributed code as of version 4.1.0.

Example of a small-residual problem

```
t <- -10:10
y <- 100/(1+.1*exp(-0.51*t))
lform<-y~a/(1+b*exp(-c*t))
ldata<-data.frame(t=t, y=y)
plot(t,y)
lstartbad<-c(a=1, b=1, c=1)
lstart2<-c(a=100, b=10, c=1)
nlsr::nlxb(lform, data=ldata, start=lstart2)
nls(lform, data=ldata, start=lstart2, trace=TRUE)
# Fix with scaleOffset
nls(lform, data=ldata, start=lstart2, trace=TRUE, control=list(scaleOffset=1.0))
sessionInfo()
```

Edited output of running this function follows:

```
> t <- -10:10
> y <- 100/(1+.1*exp(-0.51*t))
> lform<-y~a/(1+b*exp(-c*t))
> ldata<-data.frame(t=t, y=y)
> plot(t,y)
> lstartbad<-c(a=1, b=1, c=1)
> lstart2<-c(a=100, b=10, c=1)
> nlsr::nlxb(lform, data=ldata, start=lstart2)
nlsr object: x
residual sumsquares =  1.007e-19  on  21 observations
    after  13    Jacobian and  19 function evaluations
  name            coeff          SE       tstat      pval      gradient    JSingval
a                   100     2.679e-11  3.732e+12  1.863e-216  -6.425e-11       626.6
b                   0.1      3.78e-13  2.646e+11  9.125e-196  -3.393e-08       112.3
c                  0.51       6.9e-13  7.391e+11  8.494e-204   1.503e-08       2.791
> nls(lform, data=ldata, start=lstart2, trace=TRUE)
40346.    (1.08e+00): par = (100 10 1)
11622.    (2.93e+00): par = (101.47 0.49449 0.71685)
5638.0    (1.08e+01): par = (102.23 0.38062 0.52792)
642.08    (1.04e+01): par = (102.16 0.22422 0.41935)
97.712    (1.79e+01): par = (100.7 0.14774 0.45239)
22.250    (1.78e+02): par = (99.803 0.093868 0.50492)
0.025789  (1.33e+03): par = (100.01 0.10017 0.50916)
6.0571e-08 (7.96e+05): par = (100 0.1 0.51)
4.7017e-19 (1.86e+04): par = (100 0.1 0.51)
```

```
1.2440e-27 (5.71e-01): par = (100 0.1 0.51)
1.2440e-27 (5.71e-01): par = (100 0.1 0.51)
... (approx 40 lines omitted)
1.2440e-27 (5.71e-01): par = (100 0.1 0.51)
Error in nls(lform, data = ldata, start = lstart2, trace = TRUE) :
  number of iterations exceeded maximum of 50
> # Fix with scaleOffset
> nls(lform, data=ldata, start=lstart2, trace=TRUE, control=list(scaleOffset=1.0))
40346.      (1.08e+00): par = (100 10 1)
11622.      (2.91e+00): par = (101.47 0.49449 0.71685)
5638.0      (9.23e+00): par = (102.23 0.38062 0.52792)
642.08      (5.17e+00): par = (102.16 0.22422 0.41935)
97.712      (2.31e+00): par = (100.7 0.14774 0.45239)
22.250      (1.11e+00): par = (99.803 0.093868 0.50492)
0.025789    (3.79e-02): par = (100.01 0.10017 0.50916)
6.0571e-08  (5.80e-05): par = (100 0.1 0.51)
4.7017e-19  (1.62e-10): par = (100 0.1 0.51)
Nonlinear regression model
  model: y ~ a/(1 + b * exp(-c * t))
   data: ldata
     a      b      c
100.00   0.10   0.51
 residual sum-of-squares: 4.7e-19
Number of iterations to convergence: 8
Achieved convergence tolerance: 1.62e-10

Running under: Linux Mint 20.3
```

More general termination tests

The single convergence criterion of `nls()` leaves out some possibilities that could be useful for some problems. The package `nlsr` (John C Nash and Duncan Murdoch (2019)) already offers both the safeguarded relative offset test (**roffset**) as well as a **small sum of squares** test (**smallsstest**) that compares the latest evaluated sum of squared (weighted) residuals to `e4` times the initial sum of squares. The value of `e4` is computed as

```
epstol<-100*.Machine$double.eps
e4 <- epstol^4
e4
```

```
## [1] 2.430865e-55
```

We note that `nls()` uses a termination test to stop after `maxiter` "iterations". Unfortunately, for almost all iterative algorithms, the meaning of "iteration" requires careful examination of the code. we prefer to express such tests using the number of times the residuals or the jacobian have been computed and put upper limits on these. Our codes exit (terminate) when these limits are reached. Generally we prefer larger limits than the default `maxiter=50` of `nls()`, but that may simply reflect our history of dealing with more difficult problems as we are the tool-makers; users consult us when things go wrong.

Feature: Failure when Jacobian is computationally singular

This is the infamous "singular gradient" termination message of `nls()`. A Google search of

```
R nls "singular gradient"
```

gets over 4000 hits that are spread over the years. In some cases this is due to the failure of the simple finite difference approximation of the Jacobian in the `numericDeriv()` function that is a part of `nls()`. `nlsr` can use analytic derivatives, and we could import this functionality to the `nls()` code as an improvement. See below in the section **Jacobian computation**.

A common source of the issue is that the Jacobian is very close to singular for some values of the model parameters. In such cases we need to find an alternative algorithm to the Gauss-Newton iteration of `nls()`. The most common work-around is the Levenberg-Marquardt stabilization (see Marquardt (1963), Levenberg (1944), Nash (1977)). Versions of this have been implemented in packages `minpack.lm` and `nlsr`. and we have prepared experimental versions of `nls` replacements that can incorporate stabilizations. Integration of such ideas with other code structures so that all the features of `nls()` work properly has unfortunately proved difficult.

Let us consider an example which presents the infamous 'singular gradient' error message. This was posted on StackExchange. *https://stats.stackexchange.com/questions/13053/singular-gradient-error-in-nls-with-correct-starting-values*

```
reala=-3
realb=5
realc=0.5
realr=0.7
realm=1
x=1:11 #x values 11 timepoint data
#linear+exponential function
y=reala + realb*realr^(x-realm) + realc*x
testdat=data.frame(x,y)
strt<-list(a=-3, b=5, c=0.5, r=0.7, m=1)
jform<-y~a+b*r^(x-m)+c*x # Formula
linexp=try(nls(jform, data=testdat, start=strt, trace=F))
```

```
## Error in nlsModel(formula, mf, start, wts, scaleOffset = scOff, nDcentral = nDcntr) :
##   singular gradient matrix at initial parameter estimates
```

```
library(nlsr)
# Note singular values of Jacobian in rightmost column
linexp2<-try(nlxb(jform, data=testdat, start=strt, trace=F))
linexp2
```

```
## nlsr object: x
## residual sumsquares =  0  on  11 observations
##     after  1    Jacobian and  1 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a                    -3          NA          NA        NA             0       26.49
## b                     5          NA          NA        NA             0       9.615
## c                   0.5          NA          NA        NA             0       2.466
## r                   0.7          NA          NA        NA             0      0.1098
## m                     1          NA          NA        NA             0   1.398e-16
```

In the above, we see that the Jacobian is essentially singular as shown by its singular values. Note that we DISPLAY the singular values in a single column for convenience in the output. The values do NOT correspond to individual parameters, but are a property of the whole problem.

Feature: Jacobian computation

`nls()`, with the `numericDeriv()` function, computes the Jacobian as the "gradient" attribute of the residual vector. This is implemented as a mix of R and C code, but we have created a rather more compact version entirely in R. Code project. See the companion document **Jacobian Calculations for nls()** (?? need citation – currently **DerivsNLS.pdf**).

The Jacobian used by `nlsr::nlxb()` is computed from analytic expressions, while that of `nls()` will use a numerical approximation. In the most recent example above, the singularity is not due to the numerical approximation.

It should be noted that the selfStart models in the `./src/library/stats/R/zzModels.R` file provide the Jacobian in the "gradient" attribute of the "one-sided" formula that defines each model, and these Jacobians may be the analytic forms. The `nls()` function, after computing the "right hand side" or `rhs` of the residual, then checks to see if the "gradient" attribute is defined, and, if not, uses `numericDeriv` to compute a Jacobian into that attribute. This code is within the `nlsModel()` or `nlsModel.pliner()` functions. The use of analytic Jacobians almost certainly contributes to the performance of `nls()` on selfStart models.

Feature: Subsetting

`nls()` accepts an argument `subset`. Unfortunately, this acts through the mediation of `model.frame` and is not clearly obvious in the source code files `/src/library/stats/R/nls.R` and `/src/library/stats/src/nls.C`.

While the implementation of subset at the level of the call to `nls()` has a certain attractiveness, it does mean that the programmer of the solver needs to be aware of the source (and value) of objects such as the data, residuals and Jacobian. By preference, we would implement subsetting by means of zero-value weights, with observation counts (and degrees of freedom) computed via the numbers of non-zero weights. Alternatively, we would extract a working dataframe from the relevant elements in the original.

Feature: na.action

`na.action` is an argument to the `nls()` function, but it does not appear obviously in the source code, often being handled behind the scenes after referencing the option `na.action`. A useful, but possibly dated, description is given in: https://stats.idre.ucla.edu/r/faq/how-does-r-handle-missing-values/

The typical default action, which can be seen by using the command `getOption("na.action")` is `na.omit`. This option essentially presents computations with data with all observations containing any missing values (i.e. any row of a data frame with an NA) omitted. `na.exclude` does much the same for computations, but keeps the rows with NA elements so that predictions are in the correct row position. We recommend that workers actually test output to verify behaviour is as wanted.

A succinct description of the issue is given in: *https://stats.stackexchange.com/questions/492955/should-i-use-na-omit-or-na-exclude-in-a-linear-model-in-r*

> The only benefit of `na.exclude` over `na.omit` is that the former will retain the original number of rows in the data. This may be useful where you need to retain the original size of the dataset - for example it is useful when you want to compare predicted values to original values. With `na.omit` you will end up with fewer rows so you won't as easily be able to compare.

`na.pass` simply passes on data "as is", while `na.fail` will essentially stop if any missing values are present.

Feature: model frame

`model` is an argument to the `nls()` function, which is documented

> **model** logical. If true, the model frame is returned as part of the object. Default is FALSE.

Indeed, the argument only gets used when `nls()` is about to return its result object, and the element `model` is NULL unless the calling argument `model` is TRUE. (Using the same name could be confusing.) However, the model frame is used within the function code in the form of an object `mf`.

Feature: sources of data

`nls()` can be called without specifying the `data` argument. In this case, it will search in the available environments (i.e., workspaces) for suitable data objects. We do NOT like this approach, but it is "the R way". R allows users to leave many objects in the default (.GlobalEnv) workspace. Moreover, users have to actively suppress saving this workspace (`.RData`) on exit, and any such file in the path when R is launched will be loaded. The overwhelming proportion of R users in our acquaintance avoid saving the workspace because of the danger of lurking data and functions which may cause unwanted results.

Nevertheless, to provide compatible behaviour with `nls()`, we will need to ensure that equivalent behaviour is guaranteed. Furthermore, we need to test that the operation is correct.

Feature: missing start vector and self-starting models

Nonlinear estimation algorithms are almost all iterative and need a set of starting parameters. `nls()` offers a special class of modeling functions called **selfStart** models. There are a number of these in base R (see list below) and others in R packages such as CRAN package `nlraa` (Miguez (2021)), as well as the now-archived package `NRAIA`. Unfortunately, the structure of the programming of these is such that the methods by which initial parameters are computed is entangled with the particularities of the `nls()` code. Though there is a `getInitial()` function, this is not easy to use to simply compute the initial parameter estimates, in part because it may call `nls()`. Such circular references are, in our view, dangerous.

In the example below, we show how the `SSlogis` selfStart function can generate a set of initial parameters for a 3-parameter logistic curve. The form used by `SSlogis` is

$$y \sim Asym/(1 + exp((xmid - tt)/scal))$$

The example shows how these starting parameters can be transformed to those of another form of the model, namely,

$$y \sim b1/(1 + b2 * exp(-b3 * t))$$

Let us look at the actual code for `SSlogis()` in `R-devel/src/library/stats/R/zzModels.R`:

```
SSlogis <- selfStart(~ Asym/(1 + exp((xmid - input)/scal)),
    selfStart(
        function(input, Asym, xmid, scal)
        {
              .expr1 <- xmid - input
              .expr3 <- exp(.e2 <- .expr1/scal)
              .expr4 <- 1 + .expr3
              .value <- Asym/.expr4
              .actualArgs <- as.list(match.call()[c("Asym", "xmid", "scal")])
              if(all(vapply(.actualArgs, is.name, NA)))
              {
                      .expr10 <- .expr4^2
                  .grad <- array(0, c(length(.value), 3L), list(NULL, c("Asym", "xmid", "scal")))
                  .grad[, "Asym"] <- 1/.expr4
                      .grad[, "xmid"] <- - (xm <- Asym * .expr3/scal/.expr10)
                      .grad[, "scal"] <- xm * .e2
                  dimnames(.grad) <- list(NULL, .actualArgs)
                  attr(.value, "gradient") <- .grad
              }
              .value
        },
        initial = function(mCall, data, LHS, ...) {
              xy <- sortedXyData(mCall[["input"]], LHS, data)
              if(nrow(xy) < 4) {
                  stop("too few distinct input values to fit a logistic model")
              }
              z <- xy[["y"]]
              ## transform to proportion, i.e. in (0,1) :
```

```
              rng <- range(z); dz <- diff(rng)
              z <- (z - rng[1L] + 0.05 * dz)/(1.1 * dz)
              xy[["z"]] <- log(z/(1 - z))       # logit transformation
              aux <- coef(lm(x ~ z, xy))
              pars <- coef(nls(y ~ 1/(1 + exp((xmid - x)/scal)),
                               data = xy,
                               start = list(xmid = aux[[1L]], scal = aux[[2L]]),
                               algorithm = "plinear", ...))
              setNames(pars [c(".lin", "xmid", "scal")],
                       mCall[c("Asym", "xmid", "scal")])
        },
        parameters = c("Asym", "xmid", "scal"))
```

We note that the function includes analytic expressions for the Jacobian ("gradient"). These could be possibly be useful to R users, especially if documented. Moreover, we wonder why the programmers have chosen to save so many quantities in "hidden" variables, i.e., with names preceded by ".". These are then not displayed by the `ls()` command, making them difficult to access.

In the event that a selfStart model is not available, `nls()` sets all the starting parameters to 1. This is, in our view, tolerable, but could possibly be improved by using a set of values that are slightly different e.g., in the case of a model

$$y \sim a * exp(-b * x) + c * exp(-d * x)$$

it would be useful to have b and d values different so the Jacobian is not singular. Thus, some sort of sequence like 1.0, 1.1, 1.2, 1.3 for the four parameters might be better and it can be provided quite simply instead of all 1's.

```
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

NLSformula0 <- y ~ b1/(1+b2*exp(-b3*tt))
NLSformula <- y ~ SSlogis(tt, Asym, xmid, scal)
NLSformulax <- y ~ Asym/(1+exp((xmid-tt)/scal))
NLStestdata <- data.frame(y=weed, tt=tt)
s0 <- getInitial(NLSformula, NLStestdata)
print(s0)
```

```
##     Asym     xmid     scal
## 196.1862  12.4173   3.1891
```

```
# We transform the parameters for the NLSformula0 model of original specification.
s1<-list(b1=s0[1], b2=exp(s0[2]/s0[3]), b3=1/s0[3])
print(as.numeric(s1))
```

```
## [1] 196.18624  49.09163   0.31357
```

```
# No actual improvement because nls() has been already used to get the starting values,
# but we do get SEs
hobblog<-nls(NLSformula0, data=NLStestdata, start=s1)
summary(hobblog)
```

```
##
## Formula: y ~ b1/(1 + b2 * exp(-b3 * tt))
##
## Parameters:
##         Estimate Std. Error t value Pr(>|t|)
## b1.Asym 1.96e+02   1.13e+01    17.4  3.2e-08 ***
## b2.xmid 4.91e+01   1.69e+00    29.1  3.3e-10 ***
## b3.scal 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: 1.11e-06
```

```
deviance(hobblog)
```

```
## [1] 2.5873
```

```
# nls fails without selfStart -- singular gradient, even on stabilized formula
try(hobblogx<-nls(NLSformulax, data=NLStestdata))
```

```
## Warning in nls(NLSformulax, data = NLStestdata): No starting values specified for some parameters.
## Initializing 'Asym', 'xmid', 'scal' to '1.'.
## Consider specifying 'start' or using a selfStart model

## Error in nls(NLSformulax, data = NLStestdata) : singular gradient
```

```
# But Marquardt is able to get a solution easily
library(nlsr)
hobblogxx<-nlxb(NLSformulax, data=NLStestdata, start=c(Asym=1, xmid=1, scal=1))
hobblogxx
```

```
## nlsr object: x
## residual sumsquares =  2.5873  on  12 observations
##     after  23    Jacobian and  31 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## Asym             196.186        11.31      17.35  3.167e-08   4.136e-12       44.93
## xmid             12.4173       0.3346      37.11  3.716e-11   7.492e-11        15.6
## scal             3.18908       0.0698      45.69  5.768e-12   1.606e-11      0.0474
```

selfStart models in base R

The following models are provided (in file ./src/library/stats/R/zzModels.R)

```
SSasymp         - asymptotic regression model
SSasympOff      - alternate formulation of asymptotic regression model with offset
SSasympOrig     - exponential curve through the origin to an asymptote
SSbiexp         - y ~ ~ A1 * exp(-exp(lrc1)*input) + A2 * exp(-exp(lrc2) * input)
SSfol           - y ~ Dose * (exp(lKe + lKa - lCl) * (exp(-exp(lKe) * input) -
                         exp(-exp(lKa) * input))/(exp(lKa) - exp(lKe)))
SSfpl           - four parameter logistic model
SSlogis         - three parameter logistic model
SSmicmen        - Michaelis-Menten model for enzyme kinetics
SSgompertz2     - Gompertz model for growth curve data
SSweibull       - Weibull model for growth curve data
```

Strategic issues in selfStart models

Because the Gauss-Newton algorithm is rather unreliable from many starting sets of parameters, selfStart models are more than an accessory to `nls()` but a part of the infrastructure. However, creating such functions is a lot of work, and their documentation (file `./src/library/stats/man/selfStart.Rd`) is quite complicated. We believe that the focus would better be placed on getting good initial parameters, possibly with some interactive tools. That is, the emphasis should be on the `getInitial()` function, though avoiding the current calls back to `nls()`.

Issue: documentation of the results of running nls()

The output of `nls()` is an object of class "nls" which has the following structure:

nls() result output according to the documentation

```
A list of:

m            an nlsModel object incorporating the model.
data         the expression that was passed to nls as the data argument. The actual data values
               are present in the environment of the m components, e.g., environment(m$conv).
call         the matched call with several components, notably algorithm.
na.action    the "na.action" attribute (if any) of the model frame.
dataClasses  the "dataClasses" attribute (if any) of the "terms" attribute of the model frame.
model        if model = TRUE, the model frame.
weights      if weights is supplied, the weights.
convInfo     a list with convergence information.
control      the control list used, see the control argument.

There are also two deprecated items if algorithm = "port" fit only. These are
convergence (a code = 0 for convergence) and message. These are available from convInfo.
```

Example output

To illustrate, let us run the Croucher example.

```r
# Croucher example
xdata <- c(-2,-1.64,-1.33,-0.7,0,0.45,1.2,1.64,2.32,2.9)
ydata <- c(0.699369,0.700462,0.695354,1.03905,1.97389,2.41143,1.91091,0.919576,-0.730975,-1.42001)
p1<- 1; p2<-0.2; NLSstart<-list(p1=p1,p2=p2)
NLSformula <- ydata ~ p1*cos(p2*xdata) + p2*sin(p1*xdata)
NLSdata<-data.frame(xdata, ydata)
# Try full output version of nls
library(nlspkg) # use the packaged version of nls()
```

```
## Registered S3 methods overwritten by 'nlspkg':
##   method             from
##   anova.nls          stats
##   coef.nls           stats
##   confint.nls        stats
##   deviance.nls       stats
##   df.residual.nls    stats
##   fitted.nls         stats
##   formula.nls        stats
##   logLik.nls         stats
##   nobs.nls           stats
##   plot.profile.nls   stats
##   predict.nls        stats
##   print.nls          stats
##   print.summary.nls  stats
##   profile.nls        stats
##   residuals.nls      stats
##   summary.nls        stats
##   vcov.nls          stats
##   weights.nls       stats

##
## Attaching package: 'nlspkg'

## The following objects are masked from 'package:stats':
##
##     nls, nls.control, numericDeriv
```

```
result<-nls(NLSformula, data=NLSdata, start=NLSstart, model=TRUE)
# str(result) -- displays large amount of material - suppressed here
#    as it is too wide for the page
result
```

```
## Nonlinear regression model
##   model: ydata ~ p1 * cos(p2 * xdata) + p2 * sin(p1 * xdata)
##    data: NLSdata
##   p1   p2
## 1.88 0.70
##  residual sum-of-squares: 0.0538
##
## Number of iterations to convergence: 7
## Achieved convergence tolerance: 2.19e-06
```

```
ls(result) # to list the elements of the output
```

```
## [1] "call"        "control"     "convInfo"    "data"        "dataClasses"
## [6] "m"           "model"
```

```
ls(result$m) # and in particular the "m" object
```

```
##  [1] "conv"       "deviance"   "fitted"     "formula"    "getAllPars"
##  [6] "getEnv"     "getPars"    "gradient"   "incr"       "lhs"
## [11] "predict"    "resid"      "Rmat"       "setPars"    "setVarying"
## [16] "trace"
```

Concerns with content of the nls result object

The nls object contains some elements that are awkward to produce by other algorithms. Moreover, some information that would be useful is not presented obviously.

In the following, we use `result` as the returned object from `nls()`.

The `data` return element is an R symbol. To actually access the data from this element, we need to use the syntax

```
eval(parse(text=result$data))
```

However, if the call is made with `model=TRUE`, then there is a returned element `model` which contains the data, and we can list its contents using

```
ls(result$model)
```

and if there is an element called `xdata`, it can be accessed as `result$model$xdata`.

Information that is NOT in the nls result object

`nlsr::nlxb()` solves ostensibly the same problem as `nls()` but only claims to return

```
coefficients    A named vector giving the parameter values at the supposed solution.
```

`ssquares`	`The sum of squared residuals at this set of parameters.`
`resid`	`The residual vector at the returned parameters.`
`jacobian`	`The jacobian matrix (partial derivatives of residuals w.r.t. the parameters) at the returned parameters.`
`feval`	`The number of residual evaluations (sum of squares computations) used.`
`jeval`	`The number of Jacobian evaluations used.`

However, actually looking at the structure of a returned result gives a list of 11 items:

```
 $ resid       : num [1:12] 0.0119 -0.0328 0.092 0.2088 0.3926 ...
  ..- attr(*, "gradient")= num [1:12, 1:3] 0.0271 0.0367 0.0496 0.0666 0.089 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : NULL
  .. .. ..$ : chr [1:3] "Asym" "xmid" "scal"
 $ jacobian    : num [1:12, 1:3] 0.0271 0.0367 0.0496 0.0666 0.089 ...
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : NULL
  .. ..$ : chr [1:3] "Asym" "xmid" "scal"
 $ feval       : num 31
 $ jeval       : num 23
 $ coefficients: Named num [1:3] 196.19 12.42 3.19
  ..- attr(*, "names")= chr [1:3] "Asym" "xmid" "scal"
 $ ssquares    : num 2.59
 $ lower       : num [1:3] -Inf -Inf -Inf
 $ upper       : num [1:3] Inf Inf Inf
 $ maskidx     : int(0)
 $ weights     : NULL
 $ formula     :Class 'formula'  language y ~ Asym/(1 + exp((xmid - tt)/scal))
  .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
 - attr(*, "class")= chr "nlsr"
```

This is still a smaller result object than the one `nls()` returns. Moreover, `nlxb` explicitly returns the sum of squares as well as the residual vector and Jacobian. The counts of evaluations are also returned. Working on this project showed several potential updates to the `nlsr` documentation.

Weights in returned functions from nls()

The functions `resid()` (an alias for `residuals()`) and `fitted()` and `lhs()` are UNWEIGHTED. But if we return `ans` from `nls()` or `minpack.lm::nlsLM` or our new `nlsj` (interim package), then `ans$m$resid()` is WEIGHTED.

Interim output from the "port" algorithm

As the `nls()` **man** page states, when the "port" algorithm is used with the `trace` argument TRUE, the iterations display the objective function value which is 1/2 the sum of squares (or deviance). It is likely that the trace display is embedded in the Fortran of the `nlminb` routine that is called to execute the "port" algorithm, but the discrepancy is nonetheless unfortunate for users.

Failure to return best result achieved

If `nls()` reaches a point where it cannot continue but has not found a point where the relative offset convergence criterion is met, it may simply exit, especially if a "singular gradient" (singular Jacobian) is found. However, this may occur AFTER the function has made considerable progress in reducing the sum of squared residuals. An example is to be found in the `Tetra_1.R` example from the `nlsCompare` package. Here is an abbreviated version of that problem and the `nls()` output:

```
time=c( 1,  2,  3,  4,  6 , 8, 10, 12, 16)
conc = c( 0.7, 1.2, 1.4, 1.4, 1.1, 0.8, 0.6, 0.5, 0.3)
NLSdata <- data.frame(time,conc)
NLSstart <-c(lrc1=-2,lrc2=0.25,A1=150,A2=50) # a starting vector (named!)
NLSformula <-conc ~ A1*exp(-exp(lrc1)*time)+A2*exp(-exp(lrc2)*time)
tryit <- try(nls(NLSformula, data=NLSdata, start=NLSstart, trace=TRUE))
```

```
## 61216.    (3.56e+03): par = (-2 0.25 150 50)
## 2.1757    (2.23e+01): par = (-1.9991 0.31711 2.6182 -1.3668)
## 1.6211    (7.14e+00): par = (-1.9605 -2.6203 2.5753 -0.55599)
## Error in nls(NLSformula, data = NLSdata, start = NLSstart, trace = TRUE) :
##   singular gradient
```

```
print(tryit)
```

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

Note that the sum of squares has been reduced from 61216 to 1.6211, but unless `trace` is invoked, the user will not get any information about this. This would be an almost trivial change to the 'nls() function and could be useful to R users.

Feature: partially linear models and their specification

Specifying a model to a solver should, ideally, use the same syntax across solver tools. Unfortunately, R allows multiple approaches.

One obvious case is that nonlinear modeling tools are a superset of linear ones. Yet the explicit model

`y ~ a*x + b`

does not work with the linear modeling function `lm()`, which requires this model to be specified as

`y ~ x`

However, even within `nls()`, we see annoying inconsistencies. Consider the following FOUR different calling sequences for the same problem, though the second is to illustrate how one intuitive choice will not work. In this failed attempt, putting the `Asym` parameter in the model causes the `plinear` algorithm to try to add another term to the model. We believe this is unfortunate, and would like to see a consistent syntax. At the time of writing (end of August 2021) we do not have a resolution for this issue.

```
DNase1 <- subset(DNase, Run == 1)

## using a selfStart model - do not specify the starting parameters
fm1DNase1 <- nls(density ~ SSlogis(log(conc), Asym, xmid, scal), DNase1)
summary(fm1DNase1)
```

```
##
## Formula: density ~ SSlogis(log(conc), Asym, xmid, 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: 0
## Achieved convergence tolerance: 8.24e-06
```

```
## the coefficients only:
coef(fm1DNase1)
```

```
##   Asym   xmid   scal
## 2.3452 1.4831 1.0415
```

```
## including their SE, etc:
coef(summary(fm1DNase1))
```

```
##      Estimate Std. Error t value   Pr(>|t|)
## Asym   2.3452   0.078154  30.007 2.1655e-13
## xmid   1.4831   0.081353  18.230 1.2185e-10
## scal   1.0415   0.032271  32.272 8.5069e-14
```

```
## using conditional linearity - leave out the Asym parameter
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: 1.1e-06
```

```
## using conditional linearity AND Asym does NOT work

fm2aDNase1 <- try(nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(Asym=3, xmid = 0, scal = 1),
                 algorithm = "plinear",
                 trace = TRUE))
```

```
## 0.71393   (6.72e-01): par = (3 0 1 0.48462)
## 0.11356   (5.38e-01): par = (-15668537 1.5459 1.3209 -1.5264e-07)
## 0.010700  (1.73e+00): par = (-5.4588e+13 1.7343 1.0882 -4.7563e-14)
## 0.0084583 (1.51e-01): par = (1.4866e+20 1.3115 1.008 1.4747e-20)
## 0.0048018 (9.42e-01): par = (9.8031e+25 1.4766 1.0414 2.3863e-26)
## 0.0048018 (8.87e-01): par = (1.5159e+29 1.4766 1.0414 1.5432e-29)
## Error in nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)), data = DNase1,  :
##   step factor 0.000488281 reduced below 'minFactor' of 0.000976562
```

```
summary(fm2aDNase1)
```

```
##    Length     Class      Mode
##         1 try-error character
```

```
## without conditional linearity
fm3DNase1 <- nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(Asym = 3, xmid = 0, scal = 1))
summary(fm3DNase1)
```

```
##
## 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: 1.88e-06
```

```
## using Port's nl2sol algorithm
fm4DNase1 <- try(nls(density ~ Asym/(1 + exp((xmid - log(conc))/scal)),
                 data = DNase1,
                 start = list(Asym = 3, xmid = 0, scal = 1),
                 algorithm = "port"))
summary(fm4DNase1)
```

```
##
## 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
##
## Algorithm "port", convergence message: relative convergence (4)
```

Issue: code structure

The `nls()` code is structured in a way that inhibits both maintenance and improvement. In particular, the iterative setup is such that introduction of Marquardt stabilization is not easily available.

To obtain performance, a lot of the code is in C with consequent calls and returns that complicate the code. Over time, R has become much more efficient on modern computers, and the need to use compiled C and Fortran is less critical. Moreover, the burden for maintenance could be much reduced by moving code entirely to R.

Issue: code documentation for maintenance

`setPars()` – explain weaknesses. Only used by `profile.nls()`

The paucity of documentation is exacerbated by the mixed R/C/Fortran code base.

Following is an email to John Nash from Doug Bates. This is NOT a criticism of Prof. Bates work, but a reflection on how difficult it is to develop code in this subject area and to keep it maintainable. We have experienced similar loss of understanding for some of our own codes.

```
I'm afraid that I don't know the purpose of the recursive call either.
I know that I wrote the code to use a closure for the response, covariates, etc.,
but I don't recall anything like a recursive call being necessary.

If the R sources were in a git repository I might try to use `git blame`
to find out when and by whom that was written but they are in an SVN repository,
I think, and I haven't used it for a long, long time.

I don't think I will be of much help.  My R skills have atrophied to the point
where I wouldn't even know how to start exploring what is happening in the first
call as opposed to the recursive call.
```

This was in response to an email to Doug bates on Jun 29, 2021 from John Nash.

```
    Thanks.

    https://gitlab.com/nashjc/improvenls/-/blob/master/Croucher-expandednlsnoc.R
    <https://gitlab.com/nashjc/improvenls/-/blob/master/Croucher-expandednlsnoc.R>

    This has the test problem and the expanded code. Around line 367 is where we are
    scratching our heads. The function code (from nlsModel()) is in the commented lines below
    the call. This is

          # > setPars
          # function(newPars) {
          #   setPars(newPars)
          #   resid <<- .swts * (lhs - (rhs <<- getRHS())) # envir = thisEnv {2 x}
          #   dev   <<- sum(resid^2) # envir = thisEnv
          #   if(length(gr <- attr(rhs, "gradient")) == 1L) gr <- c(gr)
          #   QR <<- qr(.swts * gr) # envir = thisEnv
          #   (QR$rank < min(dim(QR$qr))) # to catch the singular gradient matrix
          # }
```

```
I'm anticipating that we will be able to set up a (possibly inefficient) code
with documentation that will be easier to follow and test, then gradually figure
out how to make it more efficient.

The equivalent from minpack.lm is

setPars = function(newPars) {
            setPars(newPars)
            assign("resid", .swts * (lhs - assign("rhs", getRHS(),
                envir = thisEnv)), envir = thisEnv)
            assign("dev", sum(resid^2), envir = thisEnv)
            assign("QR", qr(.swts * attr(rhs, "gradient")), envir = thisEnv)
            return(QR$rank < min(dim(QR$qr)))
        }

In both there is the recursive call, which must have a purpose I don't understand.
```

Feature: indexed parameters

The **man** file for `nls()` includes the following example of a situation in which parameters are indexed. It also uses the "plinear" option as an added complication. Here we use a truncated version of the example to save display space.

```
## 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).
if(! requireNamespace("MASS", quietly = TRUE)) stop("Need MASS pkg")
mm<- MASS::muscle[1:12,] # take only 1st few values of Strip
str(mm)
```

```
## 'data.frame':    12 obs. of  3 variables:
##  $ Strip : Factor w/ 21 levels "S01","S02","S03",..: 1 1 1 1 2 2 2 2 3 3 ...
##  $ Conc  : num  1 2 3 4 1 2 3 4 0.25 0.5 ...
##  $ Length: num  15.8 20.8 22.6 23.8 20.6 26.8 28.4 27 7.2 15.4 ...
```

```
mm<-droplevels(mm)
str(mm)
```

```
## 'data.frame':    12 obs. of  3 variables:
##  $ Strip : Factor w/ 3 levels "S01","S02","S03": 1 1 1 1 2 2 2 2 3 3 ...
##  $ Conc  : num  1 2 3 4 1 2 3 4 0.25 0.5 ...
##  $ Length: num  15.8 20.8 22.6 23.8 20.6 26.8 28.4 27 7.2 15.4 ...
```

```
nlev <- nlevels(mm)
withAutoprint({
  ## 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(mm, table(Strip)) # 2, 3 or 4 obs per strip
  nl <- nlevels(mm$Strip)
  ## 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)), mm,
                start = list(th = 1), algorithm = "plinear")
  summary(musc.1)
  ## 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):
  ## ?? but why use b here AND in the new formula??
  b <- coef(musc.1)

  musc.2 <- nls(Length ~ a[Strip] + b[Strip]*exp(-Conc/th), data=mm,
                start = list(a = rep(b[2], nl), b = rep(b[3], nl), th = b[1]))
  summary(musc.2)
})
```

```
## > with(mm, table(Strip))
## Strip
## S01 S02 S03
##   4   4   4
## > nl <- nlevels(mm$Strip)
## > musc.1 <- nls(Length ~ cbind(1, exp(-Conc/th)), mm, 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       0.624      0.222    2.81   0.0203 *
## .lin1   25.684      1.441   17.82  2.5e-08 ***
## .lin2  -26.631      6.147   -4.33   0.0019 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.96 on 9 degrees of freedom
##
## Number of iterations to convergence: 5
## Achieved convergence tolerance: 4.71e-07
##
## > b <- coef(musc.1)
## > musc.2 <- nls(Length ~ a[Strip] + b[Strip] * exp(-Conc/th), data = mm,
## +     start = list(a = rep(b[2], nl), b = rep(b[3], nl), th = b[1]))
## > summary(musc.2)
##
## Formula: Length ~ a[Strip] + b[Strip] * exp(-Conc/th)
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## a1  22.9277     0.5154   44.49  1.1e-07 ***
## a2  27.8606     0.5150   54.09  4.1e-08 ***
## a3  28.3426     1.0771   26.31  1.5e-06 ***
## b1 -44.1558    12.3900   -3.56    0.016 *
## b2 -43.8084    12.3148   -3.56    0.016 *
## b3 -32.8421     2.1553  -15.24  2.2e-05 ***
## th   0.5548     0.0808    6.86    0.001 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.798 on 5 degrees of freedom
##
## Number of iterations to convergence: 6
## Achieved convergence tolerance: 1.94e-06
```

Note that the answers for the parameters are NOT indexed. e.g. `coef(musc.2)` is a single level vector of parameters. We do not see `a[1], a[2], a[3]` but `a1, a2, a3`. This is because the model must integrate all the parameters because `th` is a common parameter across the index `Strip`.

We believe this structure is quite likely to cause confusion and error, and propose an alternative approach below.

Goals of our effort

Here are some of the goals we hope to accomplish.

Code rationalization and documentation

We want

- to provide a packaged version of `nls()` (call it `nlsalt`) coded entirely in R that matches the version in base R or what is packaged in `nlspkg` as described in the "PkgFromRbase" document.
- try to obtain a cleaner structure for the overall `nls()` infrastructure. By this we mean a re-factoring of the routines so they are better suited to maintenance of both the existing `nls()` methods and features as well as the new features we would like to add.
- try to explain what we do, either in comments or separate maintainer documentation. Since we are complaining about the lack of explanatory material for the current code, we feel it is incumbent on us to provide such material for our own work, and if possible for the existing code.

Rationalization of formula specifications

Below in "Consistent specifications of partially linear models" we point out that `nls()` uses a different formula specification from the default for a problem if the `plinear` algorithm is used. This is unfortunate, since the user cannot then simply add `algorithm="plinear"` to the call. Moreover, we believe it makes errors more likely. We suggest a possible approach to avoiding this issue, but have yet to cast this into working code (2021-8-19).

Rationalization of indexed models

Indexed models clearly have a place in some areas of research. However, the current approach in `nls()` is awkward. The user must use DIFFERENT formulas depending on whether the `plinear` algorithm is chosen. This seems to be related to the need for that approach to use the `lm()` function or its infrastructure, thus employing formulas that omit the parameters of models and simply give the variables or functions thereof, such as interaction terms.

Users, we believe, are much more likely to be comfortable with fully specified formulas. And such formulas are needed by nonlinear least squares functions such as `minpack.lm::nlsLM` and `nlsr::nlxb`. Thus we would like the identification of the linear parameters to be, if possible, automated. At the very least, we should be able to use a structure like

```
algorithm="plinear(parmx, thetaz)"
```

to allow for the full formula to be used with the linear parameters identified.

Provide tests

We need suitable tests in order:

- to ensure our new `nlsalt` or related packages work properly, in particular, giving results comparable to or better than the `nls()` in base R or `nlspkg`;
- to test individual solver functions to ensure they work across the range of calling mechanisms, that is, different ways of supplying inputs to the solver(s);
- to pose "silly" inputs to nonlinear least squares solvers (in R) to see if these bad input exceptions are caught by the programs.

A test runner program

?? Arkajyoti – do you want to expand?

When we have a "new" or trial solver function, we would like to know if it gives acceptable results on a range of sample problems of different types, starting parameters, input conditions, constraints, subsets, weights or other settings. Ideally we want to be able to get a summary that is easy to read and assess. For example, one approach would be to list the names of a set of tests with a red, green or yellow dot beside the name for FAILURE, SUCCESS, or "NOT APPLICABLE". In the last category would be a problem with constraints that the solver is not designed to handle.

To accomplish this, we need a suitable "runner" program that can be supplied with the name of a solver or solvers and a list of test problem cases. Problems generally have a base setup – a specification of the function to fit as a formula, some data and a default starting set of parameters. Other cases can be created by imposing bounds or mask constraints, subsets of the data, and different starts.

How to set up this "runner" and its supporting infrastructure is non-trivial. While the pieces are not as complicated as the inter-related parts of the solvers, especially `nls()`, the categorization of tests, their documentation, and the structuring to make running them straightforward and easy requires much attention to detail.

Some considerations for our test scrips:

- Is it useful to have a "base" script for each family of test problem, with numbered particular cases? That is, if we run the scripts in order, we can avoid some duplication of code and data.
- While we have developed some tags to document the test problem families and cases, we believe that such tags (essentially summary documentation) will continue to need revision as different tools and problems are included in scope of `nlsCompare`.
- Similarly, we expect that there will be ongoing review of the structure of the result files.

Outputs of the project

The project output is available in the Git repository https://github.com/nashjc/RNonlinearLS .

Formal reports or documentation

- RefactoringNLS.Rmd: this document which will become the main report of the project.
- TestsDoc.Rmd: a survey of testing tools in R. It has more general possibilities and fits into the subject of regression testing, in which case a more extensive literature review will be needed. Note that this document reflects the work in the the "Problem sets and test infrastructure" below.

Informal reports

These are documents used to discuss particular aspects of our work. These are part of the repository https://github.com/nashjc/RNonlinearLS.

- DerivsNLS.Rmd: a document to explain different ways in which Jacobian information is supplied to nonlinear least squares computation in R. File `ExDerivs.R` is a DRAFT of a script to provide examples.
- ImproveNLS.bib: a consolidated BibTex bibliography for all documents in this project, possibly with wider application to nonlinear least squares in general.
- MachineSummary.Rmd: an investigation of ways to report the characteristics and identity of machines running tests. `MachID.R` offers a suggested concise summary function to identify a particular computational system used for tests.
- VarietyInNonlinearLeastSquaresCodes.Rmd: a review of the different algorithms and the many choices in their implementation for nonlinear least squares. This is still a DRAFT at 2021-8-20.
- PkgFromRbase.Rmd: an explanation of the construction of the `nlspkg` from the code in R-base.
- WorkingDocument4ImproveNLS.Rmd: essentially a way to record what we have worked on. A project diary.

Problem sets and test infrastructure

We have several test problems and variants thereof in the `inst/scripts/` directory of the `nlsCompare` package available on Github (https://github.com/ArkaB-DS/nlsCompare). We direct the reader to that package for documentation of the test infrastructure, in particular the problems and methods files (`problems.csv` and `methods.csv`) and the various functions invoked by `run.R` to produce an output file in CSV form also.

Towards the end of the project, we have focused our attention on the `nlsCompare` package, which looks at evaluating and comparing functions for nonlinear least squares problems. We use many of the same tests as checks that are or will be built into our packages for such problems e.g., `nlsalt`. These use the `testthat` structure and may include verification of outputs that are specific to a package, such as the upper triangular matrix R of the QR decomposition that has been computed for a Jacobian. Since such an object will not be computed by all methods, testing it in `nlsCompare` makes no sense, and in that package we concentrate on the minimum sum of squares and the associated model parameters.

Code and documentation

A summary of our main results:

- nlspkg: a packaged version of the ‘nls() code from R-base. Thanks to Duncan Murdoch
- nlsalt: attempt to mirror `nls() behaviour entirely in R. This is UNFINISHED. The effort showed that the structure of the programs was difficult to follow, undocumented, and unsuited to adding improvements like the Marquardt stabilization. We were able to get a pure-R version of numericDeriv() and rework most of the functions of`nlsModel`(but not`nlsModel.plinear`). This work may continue after the project formally ends, but collaboration is likely needed with workers who have a deep knowledge of R internals.`nls-flowchart.txt`was a start at documentation of the structure of`nls()‘.
- nls-changes-for-small-residuals-in-nls-R-4.0.2.zip: collected material for the fix by JN to the relative offset convergence criterion failure when there are small residuals in problems sent to ‘`nls()`. Now part of R.
- nlsj: A refactoring of the `nls() functionality.`redesign2107.txt‘ gives some notes. This is an interim package for exploration of ideas and will NOT, as it currently is presented, become a distributed package.

- nlsralt: this is intended to be a modified version of Nash and Murdoch package `nlsr` with improvements discovered as a result of this project. At the close of this project, this is still a work in progress.

Strategic choices in a nonlinear model estimation

A key difference in approach between `nls()` and `nlsr::nlxb()`, possibly the key difference, is that `nls()` builds a large infrastructure from which the Gauss-Newton iteration can be executed and other statistical information such as profiles can be computed, while `nlxb()` returns quite limited information, and in its execution computes what is needed on an as and when basis. This follows a path that one of us (JN) established almost 50 years ago with the software that became Nash (1979), using setup, solver, and post-solution analysis phases to computation. Here, that last phase is not part of `nlxb()`, but is the domain of other functions in the `nlsr` package. `nls()` also has some similar functions, but they are much more tied into the infrastructure that is created mostly in the `nlsModel()` function.

The `nls()` approach, as implemented in the base R code and `nlspkg` leads to considerable entangling of the different functions and capabilities. This is to an extent that even after both the preparation for and carrying out of the present project, we do not feel confident to explain the code completely, nor to maintain it. However, we have made some progress toward such a goal. Part of that has been to render the code in R. In this there remain some tasks to complete, and they are possibly difficult.

Opinion for discussion

To advance the stability and maintainability of R, we believe that the functions that carry out quite advanced calculations like those achieved by `nls()` should be organized to have their interactions as orderly as possible. The aspects of `nls()` that have given us the most trouble are as follows:

- The functions that compute the residuals and jacobians are frequently computed by presuming the current data and parameters for those functions are available in an environment. As long as the correct environment is used, this provides a surprisingly short syntax to invoke the calculations.
- Weights, subsets, and various contextual controls such as that for the `na.action` that tells our code what to do with undefined or missing numbers. Again, this makes for a very simple invocation of the calculation, but the context is hidden from the user – and it can be difficult for those of us trying to maintain or improve the code to be certain we have the context correct.
- The mixing of R and C code makes for high performance, but without developer documentation leaves programmers with a lot of work to do to adjust the code. We believe in keeping code – at least a runnable reference version – in a single programming language. If necessary, by measuring ("profiling") the code, we can find bottlenecks and replace just those slower parts of the reference code.
- We believe that a structure that isolates the setup, solve, and post-solution structures for complicated functions reduces the number of objects that must be kept in appropriate alignment at any one stage in a set of calculations.
- We feel that all codes should return the best fit they have found so far, even if there are untoward conditions reached, such as a singular Jacobian. This modification could be made to `nls()` even in the short term.

Consistent specifications of partially linear models

Ideally, we believe it would be best to specify all models with a complete set of parameters. That is, the model should be written down for as it would be used, rather than for estimation. The `algorithm="plinear"` setting requires that we leave out the linear parameters. This means that we cannot simply change the `algorithm` argument without also changing the `formula`. Moreover, the partially linear parameters appear in the result object as `.lin1`, `.lin2`, etc.

A possible syntax to provide consistency would be to specify the partially linear parameters when the `plinear` option is specified. For example, we could use

```
algorithm="plinear(Asym)"
```

which would allow for the partially linear algorithm to function and the output of different `algorithm` options to be consistent.

Furthermore, we ask if it is possible for us to devise code that will detect linearity, rather than expecting the user to provide the special structure and setting. Work in this area appears to tbe quite limited, with "Automatic Linearity Detection, Report NA-13-04" (2013) being seemingly a sole contribution. Unfortunately, the mention of a "Linear and Nonlinear Discoverer" in Zhang, Cheng, and Liu (2011) is about a class of models that have almost nothing in common with the present issue. The algorithm in "Automatic Linearity Detection, Report NA-13-04" (2013) will need investigating further, but we suspect that it is complicated enough that it will not easily be adapted to an `nls()` style program. Possibly a simpler algorithm may be found.

Acknowledgement

Hans Werner Borchers has been helpful in developing the proposal for this project and in comments on this and related work.

References

"Automatic Linearity Detection, Report NA-13-04." 2013. Mathematical Institute, University of Oxford. https://core.ac.uk/download/pdf/9695588.pdf.

Bates, D. M., and D. G. Watts. 1988. *Nonlinear Regression Analysis and Its Applications*. Wiley.

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

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.

Miguez, Fernando. 2021. *nlraa: Nonlinear Regression for Agricultural Applications*. https://CRAN.R-project.org/package=nlraa.

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.

———. 1979. *Compact Numerical Methods for Computers: Linear Algebra and Function Minimisation*. Bristol: Adam Hilger.

Zhang, Hao Helen, Guang Cheng, and Yufeng Liu. 2011. "Linear or Nonlinear? Automatic Structure Discovery for Partially Linear Models." *Journal of the American Statistical Association* 106 (495): 1099–1112. http://www.jstor.org/stable/23427577.