Refactoring the `nls()` function in R

John C. Nash * Arkajyoti Bhattacharjee †

2022-6-15

Contents

Abstract

Based on work for a Google Summer of Code project "Improvements to `nls()`", we consider the features and limitations of the R function `nls()` in the context of improving and rationalizing R tools for nonlinear

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

regression.

Important considerations are the usability and maintainability of the code base that provides the functionality `nls()` claims to offer. Our work suggests that the existing code makes maintenance and improvement very difficult, with legacy applications and examples blocking some important updates. Discussion of these matters is relevant to improving R generally, as well as its nonlinear estimation tools.

The `nls()` function: strengths and shortcomings

`nls()` is the tool in base R, the primary software software distribution from the Comprehensive R Archive Network (*https://cran.r-project.org*), for estimating nonlinear statistical models. The function dates to the 1980s and the work related to Bates and Watts (1988) in S (see *https://en.wikipedia.org/wiki/S_%28programming_language%29*).

The `nls()` function has a remarkable and 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, even ones which call calculations in other programming languages
- allows data to be weighted or subset
- can estimate bound constrained parameters
- provides a mechanism for handling partially linear models
- 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
- links to a number of pre-coded ("selfStart") models

With such a range of features and long history, the code has become untidy and overly patched. It is, to our mind, essentially unmaintainable. Moreover, its underlying methods can and should be improved, as we suggest below.

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 amounted to admitting R cannot solve well-posed problems unless data is polluted with errors.

This issue can be easily resolved. The "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 projects the proposed step in the parameter vector on the gradient and estimates how much the sum of squares loss function should decrease. This estimate is divided by the current size of the loss function to avoid scale issues. When we have "converged", the estimated decrease is very small. 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.

More general termination tests

The convergence criterion above, the principal termination control 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, where `e4 <- (100*.Machine$double.eps)^4` is approximately 2.43e-55.

We note that `nls()` uses a termination test to stop after `maxiter` "iterations". Unfortunately, the meaning of "iteration" varies among programs and requires careful examination of the code. We prefer to use 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 reflect the more difficult problems we have encountered because users consult us when standard tools have given unsatisfactory results.

Feature: Failure when Jacobian is computationally singular

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

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

gets over 4000 hits that are spread over some years. This could be because the Jacobian is poorly approximated (see **Jacobian computation** below). However, it is common in nonlinear least squares computations 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)), and versions of it have been implemented in packages `minpack.lm` and `nlsr`. In our work, we prepared experimental prototypes of `nls` that incorporate stabilization, but integration with all the features of `nls()` has proved difficult.

Let us consider a 'singular gradient' example *https://stats.stackexchange.com/questions/13053/singular-gradient-error-in-nls-with-correct-starting-values*. `nlsr::nlxb()` displays the singular values of the Jacobian, which confirm that the `nls()` error message is correct, but this is not helpful in obtaining a solution.

```
reala <- -3; realb <- 5; realc <- 0.5; realr <- 0.7; realm <- 1 # underlying parameters
x <- 1:11 # x values; 11 data points
y <- reala + realb * realr^(x - realm) + realc * x # linear + exponential function
testdat <- data.frame(x, y) # save in a data frame
strt <- list(a = -3, b = 5, c = 0.5, r = 0.7, m = 1) # give programs a good start
jform <- y ~ a + b * r^(x - m) + c * x # Formula
library(nlsr)
linexp2 <- try(nlxb(jform, data = testdat, start = strt, trace = FALSE))
linexp2 # Note singular values of Jacobian in rightmost column
```

```
## 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   1.311e-16
```

```
# We get an error with nls()
linexp <- try(nls(jform, data = testdat, start = strt, trace = FALSE))
```

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

The Jacobian is essentially singular as shown by its singular values. Note that these are **displayed** by package `nlsr` in a single column in the output to provide a compact layout, but the values do **NOT** correspond to the individual parameters in whose row they appear; they 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. See *https://github.com/nashjc/RNonlinearLS/tree/main/DerivsNLS*.

As far as we can understand the logic in `nls()`, the Jacobian computation during parameter estimation is carried out entirely within the called C-language program, and the R code function `numericDeriv()`, part of `./src/library/stats/R/nls.R` in the R distribution source code. This is used to provide Jacobian information in the `nlsModel()` and `nlsModel.plinear()` functions, which are **not** exported for general use.

The Jacobian used by `nlsr::nlxb()` is, by default, computed from analytic expressions, while that of `nls()` mostly uses a numerical approximation. Some selfStart models do provide "gradient" information. `minpack.lm::nlsLM()` invokes `numericDeriv()` in its local version of `nlsModel()`, but it appears to use an internal approximate Jacobian code from the original Fortran `minpack` code, namely, `lmdif.f`. Such differences in approach can lead to different behaviour, though in our experience differences are usually minor.

- A pasture regrowth problem (Huet et al. (2004), page 1, based on Ratkowsky (1983)) has a poorly conditioned Jacobian and `nls()` fails with "singular gradient". Worse, numerical approximation to the Jacobian give the error "singular gradient matrix at initial parameter estimates" for `minpack.lm::nlsLM` so that the Marquardt stabilization is unable to take effect, while the analytic derivatives of `nlsr::nlxb` give a solution.
- Karl Schilling (private communication) provided an example where a model specified with the formula `y ~ a * (x ^ b)` causes `nlsr::nlxb` to fail because the partial derivative w.r.t. `b` is `a * (x^b * log(x))`. If there is data for which `x = 0`, this is undefined. In such cases, we observed that `nls()` and `minpack.lm::nlsLM` found a solution, though it can be debated whether such lucky accidents can be taken as an advantage.

Note that the selfStart models in the base R `./src/library/stats/R/zzModels.R` file and in package `nlraa` (Miguez (2021)) may 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.plinear()` functions. The use of analytic Jacobians almost certainly contributes to the good performance of `nls()` on selfStart models.

Feature: Weights on observations

`nls()`, `nlsr::nlxb()` and `minpack.lm::nlsLM()` all have a `weights` argument that specifies a vector of weights the same length as the number of residuals. Each residual is multiplied by the square root of the corresponding weight. The values returned by the `residuals()` function are weighted, and the `fitted()` or `predict()` function are used to compute raw residuals.

Feature: Subsetting

`nls()` accepts an argument `subset`. 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. This is overly complicated. 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 omits from computations any observations containing missing values (i.e. any row of a data frame containing an NA). `na.exclude` does much of 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 the 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* where the "Answer" states

> 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.

Our concern with `na.action` is that users may be unaware of the effects of an option setting they may not even be aware has been set. Is `na.fail` a safer default?

Feature: model frame

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

> **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 the object `mf`. We feel that users could benefit from more extensive documentation and examples of its use.

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; otherwise, 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()`, competing programs need to ensure equivalent behaviour, but users should test that the operation is as they intend.

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 (`./src/library/stats/R/zzModels.R`) and others in R packages such as CRAN package `nlraa` (Miguez (2021)), as well as the now-archived package `NRAIA`. Unfortunately, some **selfStart** codes entangle the calculation of starting values for parameters with the particulars of the `nls()` code. Though there is a `getInitial()` function, this is not easy to use to simply compute the initial parameter estimates outside of `nls()`, in part because it may call that function. Such circular references are, in our view, dangerous. Moreover, we believe that it would be helpful to have selfStart models that allow users to explicitly provide values other than those suggested for the starting parameters.

Other concerns are revealed by the example below. Here, the `SSlogis` selfStart function is used to generate a set of initial parameters for a 3-parameter logistic curve. The form used by `SSlogis` is $y \sim Asym\,/\,(1 + exp((xmid - tt)\,/\,scal))$, but we show how the starting parameters for this model can be transformed to those of another form of the model, namely, $y \sim b1\,/\,(1 + b2 * exp(-b3 * t))$.

The code for `SSlogis()` is in `./src/library/stats/R/zzModels.R`. This R function includes analytic expressions for the Jacobian ("gradient"). These could 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 be improved by using a set of values that are all slightly different, which, in the case of the example model $y \sim a * exp(-b * x) + c * exp(-d * x)$ would avoid a singular Jacobian because b and d were equal in value. A sequence like 1.0, 1.1, 1.2, 1.3 for the four parameters could be provided quite simply instead of all 1's.

Strategic issues in selfStart models

By providing starting values that are likely to be reasonable and by including Jacobian code in the selfStart model code, some of the deficiencies of the Gauss-Newton algorithm are mitigated. 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 should be placed on getting good initial parameters, that is `getInitial()` function, though avoiding the current calls back to `nls()`. Interactive tools, such as "visual fitting" (Nash and Velleman (1996)) might be worth considering.

We also note that the introduction of `scaleOffset` in R 4.1.1 to deal with the convergence test for small residual problems now requires that the `getInitial()` function have dot-arguments (`...`) in its argument list. This illustrates the entanglement of many features in `nls()`.

Issue: returned output of nls() and its documentation

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

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.
`convergence, message`	for an `algorithm = "port"` fit only, a convergence code (`0` for convergence) and message.
	To use these is *deprecated*, as they are available from `convInfo` now.

The `nls` object contains some elements that are awkward to produce by other algorithms, but some information that would be useful is not presented in a clear manner. Moreover, the complexity of the object is a challenge to users.

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`.

Let us compare the `nls()` result with that from `nlsr::nlxb()`, with ostensibly solves the same problem:

`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. The extra 5 are:

```
 $ 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"
```

The result object from `nlsr::nlxb()` is still much smaller 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. (Note that the singular values of the Jacobian are actually computed in the `print` and `summary` methods for the result.) As we made our comparisons, We noted several potential updates to the `nlsr` documentation as well as that for `nls()`.

Weights in returned functions from nls()

As already noted, the function `resid()` (an alias for `residuals()`) is WEIGHTED, as are those in the structured object `m` returned by `nls()` or `minpack.lm::nlsLM`, e.g., `ans$m$resid()`. The function `nlsModel()`, which we have had to extract from the base R code and explicitly `source()` because it is not exported to the working namespace, allows us to compute residuals for particular coefficient sets.

```
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)
wts <- 0.5^tt # simple weights
frmlogis <- weed ~ Asym/(1 + exp((xmid - tt)/scal))
Asym<-1; xmid<-1; scal<-1
nowt<-nls(weed ~ SSlogis(tt, Asym, xmid, scal)) # UNWEIGHTED
nowt
```

```
## Nonlinear regression model
##   model: weed ~ SSlogis(tt, Asym, xmid, scal)
##    data: parent.frame()
##   Asym   xmid   scal
## 196.19  12.42   3.19
##  residual sum-of-squares: 2.59
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 1.12e-06
```

```
nowt$m$resid() # This has UNWEIGHTED residual and Jacobian. Does NOT take coefficients.
```

```
##  [1] -0.011900  0.032755 -0.092030 -0.208782 -0.392634  0.057594  1.105728
##  [8] -0.715786  0.107647  0.348396 -0.652592  0.287569
## attr(,"gradient")
##           Asym     xmid    scal
##  [1,] 0.027117  -1.6229  5.8103
##  [2,] 0.036737  -2.1769  7.1111
##  [3,] 0.049596  -2.8997  8.5628
##  [4,] 0.066645  -3.8266 10.1000
##  [5,] 0.089005  -4.9881 11.6015
##  [6,] 0.117921  -6.3988 12.8762
##  [7,] 0.154635  -8.0418 13.6607
##  [8,] 0.200186  -9.8498 13.6432
##  [9,] 0.255106 -11.6901 12.5267
## [10,] 0.319083 -13.3660 10.1313
## [11,] 0.390688 -14.6444  6.5083
## [12,] 0.467334 -15.3139  2.0038
```

```
usewt <- nls(weed ~ SSlogis(tt, Asym, xmid, scal), weights=wts)
usewt
```

```
## Nonlinear regression model
##   model: weed ~ SSlogis(tt, Asym, xmid, scal)
##    data: parent.frame()
##   Asym   xmid   scal
## 199.86  12.50   3.19
##  weighted residual sum-of-squares: 0.0196
##
## Number of iterations to convergence: 2
## Achieved convergence tolerance: 8.29e-06
```

```
usewt$m$resid() # WEIGHTED. Does NOT take coefficients.
```

```
##  [1]  0.0085640  0.0324442 -0.0176652 -0.0388479 -0.0579575  0.0163623
##  [7]  0.1042380 -0.0411766  0.0052509  0.0084324 -0.0194246 -0.0024053
## attr(,"gradient")
##           Asym     xmid    scal
##  [1,] 0.026498  -1.6157  5.8228
##  [2,] 0.035901  -2.1679  7.1333
##  [3,] 0.048474  -2.8890  8.6006
##  [4,] 0.065153  -3.8149 10.1617
##  [5,] 0.087046  -4.9775 11.6983
##  [6,] 0.115387  -6.3933 13.0222
##  [7,] 0.151425  -8.0483 13.8709
##  [8,] 0.196222  -9.8787 13.9297
##  [9,] 0.250362 -11.7553 12.8918
## [10,] 0.313611 -13.4827 10.5608
## [11,] 0.384641 -14.8251  6.9663
## [12,] 0.460954 -15.5631  2.4357
```

```
source("nlsModel.R")
nmod0 <- nlsModel(frmlogis, data=weeddf, start=c(Asym=1, xmid=1, scal=1), wts=wts)
nmod0$resid() # Parameters are supplied in nlsModel() `start` above.
```

```
##  [1] 3.3998 3.2545 3.0961 2.9784 2.8438 2.7748 2.6910 2.3474 2.1724 1.9359
## [11] 1.6572 1.4214
```

```
nmod <- nlsModel(frmlogis, data=weeddf, start=coef(usewt), wts=wts)
nmod$resid()
```

```
##  [1]  0.0085640  0.0324442 -0.0176652 -0.0388479 -0.0579575  0.0163623
##  [7]  0.1042380 -0.0411766  0.0052509  0.0084324 -0.0194246 -0.0024053
```

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 factor of 2 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. Here is an abbreviated example:

```
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 almost trivial change to the `nls()` function could be useful to R users.

Feature: partially linear models and their specification

The variable projection method (Golub and Pereyra (1973), O'Leary and Rust (2013)) is generally much more effective then general approaches in finding good solutions to nonlinear least squares problems when some of the parameters appear linearly. However, setting up the calculations, that is, identifying which parameters are linear, is not a trivial task.

Moreover, specifying a model to a solver should, ideally, use the same syntax across solver tools. Unfortunately, R allows multiple approaches within different modeling tools, and within `nls()` itself. This is highlighted by the case of partially linear models.

The nonlinear modeling model specifications are, of course, a development of linear ones. Unfortunately, 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`.

Within `nls()`, consider the following FOUR different specifications for the same problem, plus an intuitive choice, labelled `fm2a`, that does 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, we do not foresee a resolution for this issue. In the example, we have NOT evaluated the commands to save space.

```
DNase1 <- subset(DNase, Run == 1) # select the data
## using a selfStart model - do not specify the starting parameters
fm1 <- nls(density ~ SSlogis(log(conc), Asym, xmid, scal), DNase1)
summary(fm1)

## using conditional linearity - leave out the Asym parameter
fm2 <- nls(density ~ 1 / (1 + exp((xmid - log(conc)) / scal)),
                 data = DNase1, start = list(xmid = 0, scal = 1),
                 algorithm = "plinear")
summary(fm2)
```

```
## without conditional linearity
fm3 <- nls(density ~ Asym / (1 + exp((xmid - log(conc)) / scal)),
                 data = DNase1,
                 start = list(Asym = 3, xmid = 0, scal = 1))
summary(fm3)

## using Port's nl2sol algorithm
fm4 <- try(nls(density ~ Asym / (1 + exp((xmid - log(conc)) / scal)),
                 data = DNase1, start = list(Asym = 3, xmid = 0, scal = 1),
                 algorithm = "port"))
summary(fm4)

## using conditional linearity AND Asym does NOT work
fm2a <- try(nls(density ~ Asym / (1 + exp((xmid - log(conc)) / scal)),
                 data = DNase1, start = list(Asym=3, xmid = 0, scal = 1),
                 algorithm = "plinear", trace = TRUE))
summary(fm2a)
```

Feature: indexed parameters

Indexed parameters – those whose names have subscripts on some element that is part of the coefficient list – are a useful idea to simplify model specification in some cases. However, they are also a focus of trouble.

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 (TRUNCATION OF EXAMPLE)
mm<-droplevels(mm) # remove unused levels after truncation
nlev <- nlevels(mm)
withAutoprint({
  ## The non linear model considered is
  ##       Length = alpha + beta*exp(-Conc/theta) + error
  ## where theta is constant. For now alpha and beta do NOT 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: 1.95e-06
```

Running the example reveals that the answers for the parameters are NOT indexed. That is, we do not see `a[1], a[2], a[3]` but `a1, a2, a3`. This is no doubt because programming for indexed parameters is extremely challenging.

Feature: bounds on parameters

There are many situations where the context of a problem constrains the values of parameters. For example, one of us was asked many years ago to estimate a model where one parameter estimate was negative. The client complained "But that is supposed to be the number of grain elevators in Saskatchewan". The number should have been a positive integer, and likely less than a few thousand.

`nls()` can impose bounds on parameters, but only if the `port` algorithm is used. Unfortunately, the manual states

> The `algorithm = "port"` code appears unfinished, and does not even check that the starting value is within the bounds. Use with caution, especially where bounds are supplied.

This is clearly unsatisfactory for general use, but does find bounded solutions. Inclusion of bounds on parameters in nonlinear least squares computations is relatively straightforward, and it is part of the default method for package `nlsr`. Package `minpack.lm` also includes provision for bounds, but the following script, which we do not run here for reasons of space, shows that it does not find a good solution for the scaled Hobbs test problem.

```
# bhobbsX.R ## bounded formula specification of problem using 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
wf <- data.frame(y = weed, tt = tt)
st <- 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
require(minpack.lm)
```

```
## Loading required package: minpack.lm
```

```
# Hobbs scaled problem with bounds, formula specification
anlxb1 <- nlxb(wmods, start = st, data = wf, lower = c(0, 0, 0), upper = c(2, 6, 3))
cat("Using nlsr::nlxb():")
```

```
## Using nlsr::nlxb():
```

```
print(anlxb1)
```

```
## residual sumsquares =  9.4726  on  12 observations
##     after  12    Jacobian and  12 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## b1                    2U           NA          NA         NA            0       23.35
## b2               4.43325           NA          NA         NA   -2.16e-07           0
## b3                    3U           NA          NA         NA            0           0
```

```
## nlsLM seems NOT to work properly with bounds
anlsLM1b <- nlsLM(wmods, start = st, data = wf, lower = c(0, 0, 0), upper = c(2, 6, 3))
cat("\n"); cat("using minpack.lm::nlsLM():")
```

```
## using minpack.lm::nlsLM():
```

```
print(anlsLM1b)
```

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

```
##    data: wf
## b1 b2 b3
##  2  6  3
##  residual sum-of-squares: 881
##
## Number of iterations to convergence: 2
## Achieved convergence tolerance: 1.49e-08
```

```
anlsb<-nls(wmods, start = st, data = wf, algorithm = "port", lower = c(0, 0, 0), upper = c(2, 6, 3))
cat("Using nls with 'port':")
```

```
## Using nls with 'port':
```

```
print(anlsb)
```

```
## Nonlinear regression model
##   model: y ~ 100 * b1/(1 + 10 * b2 * exp(-0.1 * b3 * tt))
##    data: wf
##   b1   b2   b3
## 2.00 4.43 3.00
##  residual sum-of-squares: 9.47
##
## Algorithm "port", convergence message: both X-convergence and relative convergence (5)
```

Philosophical considerations

Bounds on parameters raise some interesting and difficult questions about how uncertainty in parameter estimates should be computed or reported. That is, the traditional "standard errors" are generally taken to imply symmetric intervals about the point estimate in which the parameter may be expected to be found with some probability under certain assumptions. Bounds change those assumptions. Hence, `nlsr::nlxb()` does not compute standard errors nor their derived statistics when bounds are active.

Issue: code structure and documentation for maintenance

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 when the code was written, a large portion was written in C with consequent calls and returns that severely 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 reduced by moving code entirely to R.

While R and its packages are generally very well documented for usage, that documentation rarely extends as far as it might for code maintenance. The paucity of such documentation is exacerbated by the mixed R/C/Fortran code base, where seemingly trivial differences in things like representation of numbers or base index for arrays lead to traps for unwary programmers.

Goals and progress of our effort to improve `nls()`

Code rationalization and documentation

When we started *Improvements to nls()* for the 2021 Google Summer of Code we wanted:

- to provide a packaged version of `nls()` (call it `nlsalt`) coded entirely in R that matches the features in base R or what is packaged in `nlspkg` as described in Nash and Bhattacharjee (2022). The R version can either serve as the operational code or as a reference version to facilitate the maintenance of C, Fortran, or other versions of the software.

- to streamline 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 new features we or others would like to add.
- to explain what we do, either in comments, examples, 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.

These goals echo themes in the recent discussion by Vidoni (2021) and accompanying commentary.

Rationalization of formula specifications

We have seen that `nls()` uses a different formula specification from the default if the `plinear` algorithm is used. This is unfortunate, since the user cannot then simply add `algorithm="plinear"` to the standard call, making errors more likely.

Nonlinear models, apart from those using the "plinear" algorithm, need all the parameters specified. That is, the model formula should be written down as it would be computed. Thus a linear model (which is valid input for nonlinear estimation) should be specified `y ~ a * x + b` rather than `y ~ x`. Moreover, we want to avoid the partially linear parameters appearing in the result object as `.lin1`, `.lin2`, etc., where the actual position of parameters in the model is not explicit. We are unclear why these parameters begin with a dot, that is, that they have the "hidden" indicator.

A possible workaround to allow consistency would be to specify the partially linear parameters when the `plinear` option is specified. For example, we could use `algorithm="plinear(Asym)"` while keeping the general model formula from the default specification. This would allow for the output of different `algorithm` options to be consistent. However, we have not yet tried to code such a change, and welcome discussion from those working with other packages using model formulas.

A further complication of model specification in R is that some formulas require the user to surround a term with `I()` to inhibit interpretation of arithmetic operators. We would prefer that formulas be usable without this complication.

Rationalization of indexed models

Indexed models clearly have a place in some areas of research. We need to be able to process models such as `Length ~ a[Strip] + b[Strip] * exp(-Conc / th)` where `Strip` is the index. Data `Length` and `Conc` are available for all observations and parameter `th` applies to every fitted point, but `a[Strip]` and `b[Strip]` apply only to data indexed, which could be an experimental run number, or some other categorization.

Given that `[ ]` are the R method to index arrays, their presence in a model formula signals indexing. The question is how to perform the correct calculations.

Our view is that inclusion of the indexing **within** `nls()` or similar tools introduces complexity that could be avoided by a suitable wrapper function. The modeling formula must be expanded to encompass all the index cases, likely with indexed parameters renamed e.g., `a[2]` becomes `a2` as at present with `nls()`. However, a wrapper could process output to reconstitute the indexing of parameters.

Streamlining code

We are pessimistic that the overall structure of `nls()` can be streamlined due to the entanglement of so many features with the complicated mix of R, C and Fortran. Indeed, despite digging into the code over some months, we do not feel confident that we sufficiently understand it nor that we could maintain it. On the other hand, we do believe equivalent functionality can be built, but with at least some differences in how the features will be accessed.

Tests and use-case examples

Maintainers of packages need suitable tests and use-case examples in order:

- to ensure packages work properly, in particular, giving results comparable to or better than the functions they are to replace;
- 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 see if these bad inputs are caught by the programs.

Such goals align with the aims of **unit testing** (e.g., *https://towardsdatascience.com/unit-testing-in-r-68ab9cc8d211*, Wickham (2011), Wickham et al. (2021) and the conventional R package testing tools).

A testing package: NLSCompare

One of us has developed a working prototype package at *https://github.com/ArkaB-DS/nlsCompare* . A primary design objective of this is to allow the summarisation of multiple tests in a compact output. The prototype has a vignette to illustrate its use.

Documentation and resources

In our investigation, we built several resources, which are now part of the repository *https://github.com/nashjc/RNonlinearLS/*. Particular items are:

- A BibTex bibliography for use with all documents in this project, but which has wider application to nonlinear least squares projects in general (*https://github.com/nashjc/RNonlinearLS/blob/main/BibSupport/ImproveNLS.bib*).
- `MachID.R` offers a suggested concise summary function to identify a particular computational system used for tests. A discussion of how it was built and the resources needed across platforms is given in at *https://github.com/nashjc/RNonlinearLS/tree/main/MachineSummary*.
- Nash and Bhattacharjee (2022) is an explanation of the construction of the `nlspkg` from the `nls()` code in R-base.
- As the 2021 Summer of Code period was ending, one of us (JN) was invited to prepare a review of optimization in R. Ideas from the present work have been instrumental in the creation of Nash (2022).

Strategic choices in nonlinear model estimation

It is possible that `nls()` will remain more or less as it has been for the past two decades. Given its importance to R, the focus of discussion should be the measures needed to secure its continued operation for legacy purposes and how that may be approached. We welcome an opportunity to participate in such conversations.

To advance the stability and maintainability of R, we believe the program objects (R functions) that are created by tools such as `nls()` should have minimal cross-linkages and side-effects. The aspects of `nls()` that have given us the most trouble:

- The functions that compute the residuals and jacobians often presume the data and parameters needed are available in a particular environment. As long as the correct environment is used, this provides a compact syntax to invoke the calculations. The danger is that the wrong data is accessed if the internal search finds a valid name that is not the object we want.
- Weights, subsets, and various contextual controls (such as that for the `na.action` option) are similarly taken from the first available source, making for a very simple invocation of calculations, but the context is hidden from the user. Furthermore, 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, C and Fortran code was necessary for computational performance in the past. Lacking decent developer documentation, programmers now face a lot of work to fix or improve code. We believe in having at least a working reference version of code in a single programming language. If necessary, by measuring ("profiling") the code, we can find bottlenecks and substitute for those slower parts of the reference code.
- A design that isolates the setup, solution, and post-solution parts of for complicated calculations reduces the number of objects that must be kept in alignment. On the other hand, it may mean users have to add more commands to complete their computations.
- All iterative codes should return the best solution they have found so far, even if there are untoward conditions reached, such as a singular Jacobian. This modification could be made to `nls()` in the short term, correcting the issue identified above under "Failure to return best result achieved".
- Given the existence of examples of good practices such as analytic derivatives, stabilized solution of Gauss-Newton equations and bounds-constrained parameters, base R tools should be moving to incorporate them.

We anticipate that tools like `nlsr` will become more attractive as users discover the increased robustness and general capabilities, along with the more recent documentation. From a design point of view, a key difference in approach between `nls()` and `nlsr::nlxb()` is that `nls()` builds a large infrastructure, especially the `nlsModel()` function, 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 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), separating setup, solver, and post-solution analysis phases of computations.

Acknowledgement

Hans Werner Borchers has been helpful in developing the proposal for this project and in comments on this and related work. Heather Turner co-mentored the project and helped guide the progress of the work. Exchanges with Fernando Miguez helped to clarify aspects of selfStart models and improve package `nlsr` in that respect.

References

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.

Golub, G. H., and V. Pereyra. 1973. "The Differentiation of Pseudo-Inverses and Nonlinear Least Squares Problems Whose Variables Separate." *SIAM Journal of Numerical Analysis* 10 (2): 413–32.

Huet, S., A. Bouvier, M.-A. Poursat, and E. Jolivet. 2004. *Statistical Tools for Nonlinear Regression: A Practical Guide with S-PLUS Examples, 2nd Edition.* Berlin & New York: Springer-Verlag.

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.

———. 2022. "Function minimization and nonlinear least squares in R." *WIREs Computational Statistics*, no. e1580. https://doi.org/https://doi.org/10.1002/wics.1580.

Nash, John C., and Arkajyoti Bhattacharjee. 2022. "Making a Package from Base R Files." *R-Bloggers*, January. https://www.r-bloggers.com/2022/01/making-a-package-from-base-r-files/.

Nash, John C., and Paul Velleman. 1996. "Nonlinear Estimation Combining Visual Fitting with Optimization Methods." In *Proceedings of the Section on Physical and Engineering Sciences of the American Statistical Association*, 256–61. American Statistical Association.

O'Leary, Dianne P., and Bert W. Rust. 2013. "Variable Projection for Nonlinear Least Squares Problems." *Computational Optimization and Applications* 54 (3): 579–93.

Ratkowsky, David A. 1983. *Nonlinear Regression Modeling: A Unified Practical Approach*. New York; Basel: Marcel Dekker Inc.

Vidoni, Melina. 2021. "Software Engineering and R Programming: A Call for Research." *The R Journal* 13 (2): 6–14. https://doi.org/10.32614/RJ-2021-108.

Wickham, Hadley. 2011. "testthat: Get Started with Testing." *The R Journal* 3: 5–10. https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf.

Wickham, Hadley, Jim Hester, Winston Chang, and Jennifer Bryan. 2021. *devtools: Tools to Make Developing R Packages Easier*. https://CRAN.R-project.org/package=devtools.