Referee report of "A Comparison of R Tools for Nonlinear Least Squares Modeling".

ID: 2023-15

General

This is a long paper that covers many topics related to nonlinear least squares fitting with R. This mixture of many topics can be confusing and I think the paper would benefit from a clearer definition of its purpose and focus. Once a clearer focus is established it will be easier to decide which topics are relevant to that purpose.

The paper's title indicates that the purpose is to compare and contrast facilities in base R, meaning the `nls` function and related facilities in the `stats` package, and several external packages for nonlinear least-squares modeling that are available on CRAN. There is a listing of packages related to nonlinear least squares on page 2 but the majority of the discussion is focussed on `nls`, in the `stats` package that is part of base R, and the `nlsr` package (Nash and Murdoch 2023), written by this paper's authors and others (on CRAN this package is said to be the work of four authors but only two of them are cited in the paper).

There are also indications that the paper is about *replacing* `nls` in base R with functions from `nlsr`. The abstract begins by citing the Google Summer of Code project "Improvements to `nls()`" and the first sentence in the "**Principal messages**"" section indicates that the purpose is to create a refactored `nls()` based on the work described here. Toward the end of the paper, the section on "**Future of nonlinear model estimation in R**" also indicates that the goal is to enhance or replace `nls()` and related parts of base R.

So is the purpose of the paper:

- To introduce the `nlsr` package and describe its capabilities?
- To compare and contrast several available packages for nonlinear least-squares?
- To advocate that the authors' code replace `nls` in base R?

I would recommend concentrating on the first objective — introducing the `nlsr` package and demonstrating the advantages that it provides relative to `nls` — and I will address my comments to that purpose. I hope this approach will be of use to the authors and the editor.

It is certainly appropriate when describing the `nlsr` package to contrast it with facilities like `nls()` in base R. To help with this I will provide some background on the development of `nls()` and the `stats` package in general.

Detailed comments

nls is S and in R

Although `nls()` has been part of R for over 25 years, it originated much earlier in the "Statistical Models in S" project, documented in Chambers and Hastie (1992), and based on work starting in the mid-1980's. The purpose was to create a unified set of tools for fitting statistical models, including nonlinear models with a scalar objective, such as a log-likelihood, or as a nonlinear regression model. Many capabilities for R, including S3 classes and methods, the formula language for linear predictors, and model frames and model matrices, were based directly on the results of this project in S. (As described by Ross Ihaka, one of the two original developers of R, their goal was to produce a language that was "not unlike S" so the resemblance was not accidental.)

What was called QPE (quantitative programming environment), then “The New S Language”, then “S3” represented a huge step forward for nonlinear modeling because it could represent model functions as expressions (i.e. abstract syntax trees) that could be manipulated in the system itself. This allowed for symbolic differentiation.

The original implementation of the `deriv` function for symbolic differentiation was written in S. When porting `deriv` to R in the mid-1990’s Ross Ihaka moved quite a bit of the functionality to C code because it was comparatively easy to do so and, on the hardware of the day, the C implementation provided a considerable speed advantage. The speed of the differentiation itself isn’t usually a problem - it’s the common-sub-expression elimination that can be the bottleneck.

This legacy is also the reason that the Jacobian matrix is the `"gradient"` attribute of the returned value of the model function. For a nonlinear least squares problem, where the model function returns a vector, this deriv attribute should be called `"Jacobian"` but the version for the scalar objective, for which the vector of partial derivatives is the gradient, was written first. Because `deriv` only has access to the expression and not to the model frame it is not possible to determine if the value will be a scalar or a vector and the name of the attribute was left as `"gradient"`. A further complication is that numerical values in S and R are always stored as vectors, with the understanding that a scalar is represented as a vector of length 1. Even if it were desirable to distinguish between scalars and vectors, internally it is impossible to determine if a vector of length 1 is intended to be a scalar or to be a vector that just happens to have length 1.

The legacy is also the reason that initial data manipulation, such as evaluating a `subset` or `na.action`, is performed by non-standard evaluation of a `model.frame`. The non-standard evaluation means that, within the `nls` function evaluation, the code accesses its own call (with `match.call`) and re-writes the function name to be `stats::model.frame` then evaluates the rewritten call (around line 580 of nls.R). Handling all the special cases is complicated but the purpose is so that arguments like `subset` or `na.action` and conversion of formulas to model frames to model matrices are handled consistently across the modeling functions in the `stats` package and without code duplication.

Thomas Lumley has called this “replace the function name in my own call” approach the “standard, non-standard evaluation idiom” in R.

Another aspect of this legacy is the expected behavior of `print`, `summary`, `plot`, etc. applied to an object representing a fitted model. Peculiarities like `summary` producing longer descriptions than `print` or the choice of printing the sum of squared residuals or the estimated standard deviation of the residuals are for consistency with the other model output.

It is also why the extractor for the parameter values is `coef`, even though the parameters in a nonlinear regression model are not necessarily coefficients, and why they are shown in the summary using `print.coefmat`.

These choices, which allow users to transfer experience from one modelling situation to another, are neither arbitrary nor trivial.

Function closure with a shared environment

One of the major differences between the S and R implementations of `nls` is the use of function closures in R to allow several functions in the `m` field of an `nls` model (see below for an example) shared access to parameter values, covariate values, the response vector and various pieces of auxillary information.

In R if a function is defined during the evaluation of another function its “closure” includes the evaluation environment of its parent. If several functions are defined and returned, say in a `list`, then they have access to this common environment. This shared environment is where parameter updates and subsequent evaluation of the model function, the Jacobian and various decompositions occur.

Self-starting models in S and R

Self-starting models were developed for `nlme` package for S, Pinheiro and Bates (2000), to automate the fitting of nonlinear regression models to repeated measures data collected on several observational units, e.g. several different subjects. Rather than requiring the analyst to provide starting values for each subject's data, the steps of formulating and refining such starting values were encapsulated in functions. These functions also used symbolic derivatives to provide Jacobian evaluations and frequently fit a reduced form of the model using the `plinear` algorithm to refine the starting values, resulting in the "starting values" being the actual parameter estimates. These are fed into the default algorithm to produce a summary but with the understanding that the default algorithm will converge at the first evaluation (i.e. 0 iterative steps) if the convergence criterion is the relative offset. The point here is that the relative offset criterion depends only on the current parameter values and not on changes in the parameter values so there is only one additional evaluation required after the plinear algorithm converges.

For the illustrative example starting at the bottom of page 2 of this paper the `SSlogis` self-starting model can be used.

```r
weeddf <- data.frame(
    tt = 1:12,
    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
    )
)
summary(hobbsm1 <- nls(weed ~ SSlogis(tt, Asym, xmid, scal), weeddf))
```

```

Formula: weed ~ SSlogis(tt, Asym, xmid, scal)

Parameters:
     Estimate Std. Error t value Pr(>|t|)    
Asym 196.1862    11.3069   17.35 3.17e-08 ***
xmid  12.4173     0.3346   37.11 3.72e-11 ***
scal   3.1891     0.0698   45.69 5.77e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5362 on 9 degrees of freedom

Number of iterations to convergence: 0 
Achieved convergence tolerance: 1.118e-06
```

```r
str(hobbsm1)
```

```
List of 6
 $ m          :List of 16
  ..$ resid     :function ()  
  ..$ fitted    :function ()  
  ..$ formula   :function ()  
  ..$ deviance  :function ()  
  ..$ lhs       :function ()  
  ..$ gradient  :function ()  
  ..$ conv      :function ()  
  ..$ incr      :function ()  
  ..$ setVarying:function (vary = rep_len(TRUE, np))  
  ..$ setPars   :function (newPars)  
  ..$ getPars   :function ()  
  ..$ getAllPars:function ()  
  ..$ getEnv    :function ()  
  ..$ trace     :function ()
  ..$ Rmat      :function ()
  ..$ predict   :function (newdata = list(), qr = FALSE)
  ..- attr(*, "class")= chr "nlsModel"
 $ convInfo   :List of 5
  ..$ isConv     : logi TRUE
  ..$ finIter    : int 0
  ..$ finTol     : num 1.12e-06
  ..$ stopCode   : int 0
  ..$ stopMessage: chr "converged"
 $ data       : symbol weeddf
 $ call       : language nls(formula = weed ~ SSlogis(tt, Asym, xmid, scal), data = weeddf, algorith
 $ dataClasses: Named chr "numeric"
  ..- attr(*, "names")= chr "tt"
 $ control    :List of 7
  ..$ maxiter    : num 50
  ..$ tol        : num 1e-05
  ..$ minFactor  : num 0.000977
  ..$ printEval  : logi FALSE
  ..$ warnOnly   : logi FALSE
  ..$ scaleOffset: num 0
  ..$ nDcentral  : logi FALSE
 - attr(*, "class")= chr "nls"
```

This only takes a few milliseconds on a laptop computer

```
microbenchmark::microbenchmark(nls(weed ~ SSlogis(tt, Asym, xmid, scal), weeddf), times=100L)
```

```
Unit: milliseconds
                                              expr      min       lq     mean
 nls(weed ~ SSlogis(tt, Asym, xmid, scal), weeddf) 3.483984 3.624459 4.599596
  median       uq      max neval
 3.92275 5.035838 13.76178   100
```

although it turns out that the `SSlogisJN` formulation is slightly faster in this case

```
summary(hobbsm2 <- nls(weed ~ nlsr::SSlogisJN(tt, Asym, xmid, scal), weeddf))
```

```


Formula: weed ~ nlsr::SSlogisJN(tt, Asym, xmid, scal)

Parameters:
     Estimate Std. Error t value Pr(>|t|)
Asym 196.1862    11.3069   17.35 3.17e-08 ***
xmid  12.4173     0.3346   37.11 3.72e-11 ***
scal   3.1891     0.0698   45.69 5.77e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5362 on 9 degrees of freedom

Number of iterations to convergence: 3
Achieved convergence tolerance: 2.419e-06
```

```
microbenchmark::microbenchmark(nls(weed ~ nlsr::SSlogisJN(tt, Asym, xmid, scal), weeddf), times=1
Unit: milliseconds
                                                         expr      min       lq
 nls(weed ~ nlsr::SSlogisJN(tt, Asym, xmid, scal), weeddf) 1.960326 2.069342
     mean   median       uq      max neval
 2.492946 2.184441 2.554391 11.43154   100
```

Some of the discussion in the "**Jacobian code in selfStart models**" section (p. 9) of this paper claims that the plinear fit followed by a fit using the default approach would be difficult to maintain and could be very slow but that has not been the case. It is known (and, in fact, stated in the documentation for `SSlogis`) that the call to the default nls algorithm will always declare convergence immediately.

Also in that section it is claimed that `deriv` creates "hidden variables" because the default for the `tag` argument to `deriv` is `.expr`. This tag is only used in bindings within the evaluation environment of the model function, which is never visible to the user unless they are inspecting the function evaluation with a symbolic debugger. The only reason for the default tag starting with `"."` is to avoid conflicts with parameter or covariate names in the model expression. But it is only the default.

Recommendations

As stated in the "General" section, I think the purpose of the paper should be clarified. If the purpose is to introduce the `nlsr` package and contrast it with the existing implementation of `nls` in the `stats` package, then it might be helpful to understand why `nls` was implemented the way it was.

I hope that this unconventional referee's report is of use to the authors and the editor.

References

Chambers, John M., and Trevor J. Hastie, eds. 1992. *Statistical Models in S*. Routledge. https://doi.org/10.1201/9780203738535.

Nash, John C, and Duncan Murdoch. 2023. *Nlsr: Functions for Nonlinear Least Squares Solutions - Updated 2022*. https://CRAN.R-project.org/package=nlsr.

Pinheiro, José, and Douglas Bates. 2000. *Mixed-Effects Models in S and S-PLUS*. Springer New York.