Refactoring the nls() function in R

John C Nash, retired professor, University of Ottawa
Arkajyoti Bhattacharjee, Indian Institute of Technology, Kanpur

2021-7-17

Contents

Abstract

This article reports the particular activities of our Google Summer of Code project "Improvements to nls()" that relate to R code for that function, which is intended for the estimation of models written as a formula that has at least one parameter that is not estimable via solving a 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.

Our work has not fully addressed all the issues that we would like to see resolved, but we believe we have made sufficient progress to demonstrate that there are worthwhile improvements that can be made to the R function `nls()`. 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.

The existing nls() function: strenghts and shortcomings

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

Issue: Convergence and termination tests

Within the standard documentation (**man**ual or ".Rd" file) `nls()` warns

> **The default settings of nls generally fail on artificial "zero-residual" data problems.**
>
> The nls function uses a relative-offset convergence criterion that compares the numerical imprecision at the current parameter estimates to the residual sum-of-squares. 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. To avoid a zero-divide in computing the convergence testing value, a positive constant scaleOffset should be added to the denominator sum-of-squares; it is set in control; this does not yet apply to algorithm = "port."

It turns out that this issue can be quite easily resolved. The key "convergence test" – more properly a "termination test" for the **program** rather than testing for convergence of the underlying **algorithm** – is the Relative Offset Convergence Criterion (see Douglas M. Bates and Watts (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. To avoid scale issues, we use the current size of the loss function as a measure and divide by it. When we have "converged," the estimated decrease is very small, as usually is its ratio to the sum of squares. However, in some cases we have the possibility of an exact fit and the sum of squares is (almost) zero and we get the possibility of a zero-divide failure.

The issue is easily resolved by adding a small quantity to the loss function. To preserve legacy behaviour, in 2021, one of us (JN) proposed that `nls.control()` have an additional parameter `scaleOffset` with a default value of zero for legacy behaviour. 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**.

We are pleased to report that this improvement is in the R-devel distributed code at time of writing and will migrate to the base R distribution when updated.

Example of a small-residual problem

```
rm(list=ls())
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:

```
> rm(list=ls())
> 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)
> 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
# Note that this has succeeded. The test in nlsr recognizes small residual problems.
> 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)
1.2440e-27 (5.71e-01): 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)
1.2440e-27 (5.71e-01): 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)
1.2440e-27 (5.71e-01): 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)
```

```
1.2440e-27 (5.71e-01): 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)
1.2440e-27 (5.71e-01): 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)
1.2440e-27 (5.71e-01): 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)
1.2440e-27 (5.71e-01): 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)
1.2440e-27 (5.71e-01): 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)
1.2440e-27 (5.71e-01): 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)
1.2440e-27 (5.71e-01): 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)
1.2440e-27 (5.71e-01): 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)
1.2440e-27 (5.71e-01): 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)
1.2440e-27 (5.71e-01): 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)
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

> 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
```

```
> sessionInfo()
R version 4.1.0 (2021-05-18)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Linux Mint 20.2
```

More general termination tests

The single convergence criterion of `nls()` leaves out some possibilities that could be useful for some problems. The package `nlsr` (Nash and 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 a very small multiple of the initial sum of squares. The multiple uses a control setting `offset` which defaults to 100.0 and we compute the 4th power of the machine epsilon times this offset.

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

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

We do note that `nls()` stops after `maxiter` "iterations." However, for almost all iterative algorithms, the meaning of "iteration" requires careful examination of the code. Instead, we prefer to record 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 when things go wrong.

Issue: Failure when Jacobian is computationally singular

This is the infamous "singular gradient" termination. 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 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 can import this functionality to the `nls()` code as an improvement. See below in the section **Jacobian computation**.

However, the more 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 (Marquardt (1963), Levenberg1944, jn77ima). Versions of this have been implemented in packages `minpack.lm` and `nlsr`. and we have preliminary versions of an `nls` replacement that can incorporate a version of the Levenberg-Marquardt stabilization. (There are some issues of integration with other code structures and of complexity of the computations that suggest we should use a simplified LM stabilization.)

Issue: Jacobian computation

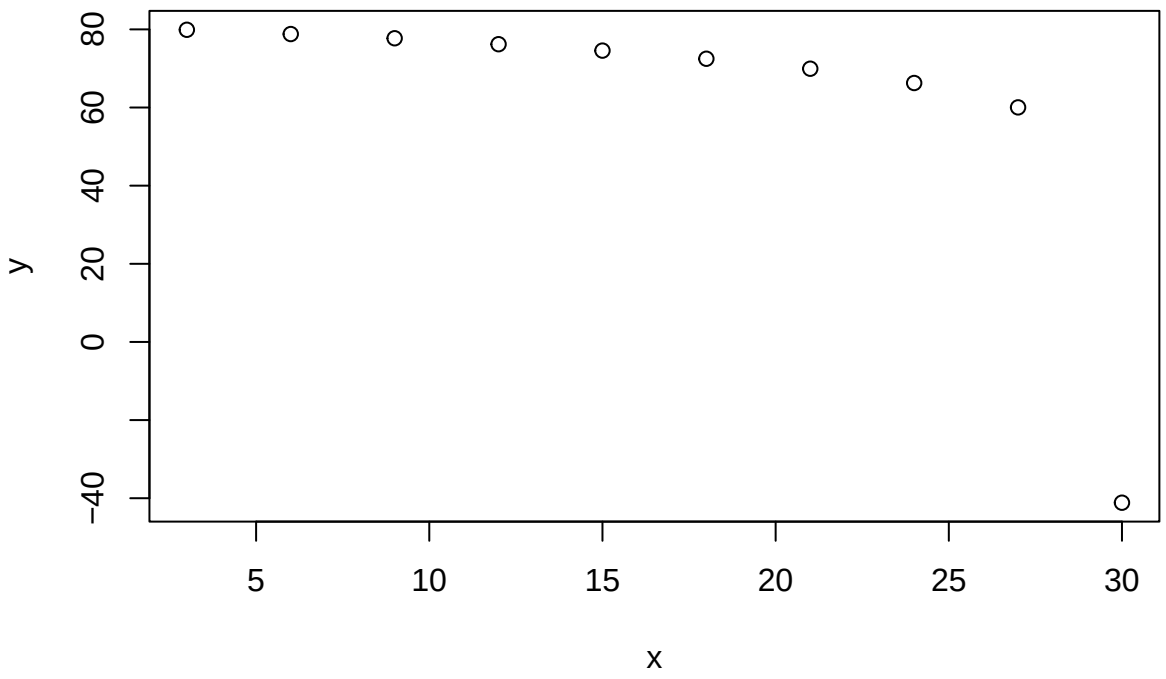
`nls()`, with the `numericDeriv()` function, computes the Jacobian as the "gradient" attribute of the residual vector. This is implemented as a mix of R and C code, but we have created a rather more compact version entirely in R in this Google Summer of Code project. See the document **DerivsNLS.pdf**.

```
# File: badJ2.R
# A problem illustrating poor numeric Jacobian
form<-y ~ 10*a*(8+b*log(1-0.049*c*x)) # the model formula
# This model uses log near a small argument, which skirts the dangerous
# value of 0. The parameters a, b, c could all be 1 "safely" as a start.
x<-3*(1:10) # define x
np<-length(x)
```

```
a<-1.01
b<-.9
eps<-1e-6
c<-1/(max(x)*.049)-eps
cat("c =",c,"\n")
```

```
## c = 0.6802711
```

```
y <- 10*a*(8+b*log(1-0.049*c*x))+0.2*runif(np) # compute a y
df<-data.frame(x=x, y=y)
plot(x,y) # for information
```




```
st<-c(a=1, b=1,c=c) # set the "default" starting vector
n0<-try(nls(form, start=st, data=df)) # and watch the fun as this fails.
summary(n0)
```

```
##
## Formula: y ~ 10 * a * (8 + b * log(1 - 0.049 * c * x))
##
## Parameters:
##    Estimate Std. Error   t value Pr(>|t|)
## a 1.011e+00  3.503e-04 2.885e+03   <2e-16 ***
## b 8.957e-01  2.258e-03 3.967e+02   <2e-16 ***
## c 6.803e-01  3.379e-08 2.013e+07   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.05131 on 7 degrees of freedom
##
## Number of iterations to convergence: 3
## Achieved convergence tolerance: 2.94e-06
```

```
library(nlsr) # but this will work
n1<-nlxb(form, start=st, data=df)
```

```
n1
```

```
## nlsr object: x
## residual sumsquares =  0.018431  on  10 observations
##     after  4    Jacobian and  5 function evaluations
##   name            coeff          SE       tstat      pval      gradient    JSingval
## a                1.01061     0.0003503       2885  1.588e-22  -4.307e-10     9488029
## b               0.895652      0.002258      396.7  1.709e-16  -1.458e-09       218.6
## c               0.680271     3.397e-08   20024024  2.046e-49  -6.265e-05       22.58
```

```
coef(n1)-coef(n0)
```

```
##          a          b          c
## 3.7625e-11 3.1792e-10 3.7192e-14
## attr(,"pkgname")
## [1] "nlsr"
```

```
jmod<-model2rjfun(form, pvec=st,data=data.frame(x=x, y=y)) # extract the model
Jatst<-jmod(st) # compute this at the start from package nlsr
Jatst<-attr(Jatst,"gradient") # and extract the Jacobian
#
# Now try to compute Jacobian produced by nls()
env<-environment(form) # We need the environment of the formula
eform<-eval(form, envir=env) # and the evaluated expression
localdata<-list2env(as.list(st), parent=env)
jnlsatst<-numericDeriv(form[[3L]], theta=names(st), rho=localdata)
Jnls<-attr(jnlsatst,"gradient")
Jnls # from nls()
```

```
##          [,1]      [,2]        [,3]
##  [1,]  78.946   -1.0536 -1.6333e+00
##  [2,]  77.769   -2.2314 -3.6750e+00
##  [3,]  76.433   -3.5667 -6.3000e+00
##  [4,]  74.892   -5.1082 -9.8000e+00
##  [5,]  73.069   -6.9315 -1.4700e+01
##  [6,]  70.837   -9.1629 -2.2050e+01
##  [7,]  67.960  -12.0397 -3.4300e+01
##  [8,]  63.906  -16.0943 -5.8800e+01
##  [9,]  56.974  -23.0257 -1.3230e+02
## [10,] -54.302 -134.3025 -1.0051e+07
```

```
Jatst # from nlsr -- analytic derivative
```

```
##             a         b           c
##  [1,]  78.946   -1.0536 -1.6333e+00
##  [2,]  77.769   -2.2314 -3.6750e+00
##  [3,]  76.433   -3.5667 -6.3000e+00
##  [4,]  74.892   -5.1082 -9.8000e+00
##  [5,]  73.069   -6.9315 -1.4700e+01
##  [6,]  70.837   -9.1629 -2.2050e+01
##  [7,]  67.960  -12.0397 -3.4300e+01
##  [8,]  63.906  -16.0943 -5.8800e+01
##  [9,]  56.974  -23.0257 -1.3230e+02
## [10,] -54.302 -134.3025 -1.0000e+07
```

```
max(abs(Jnls-Jatst))
```

```
## [1] 51029
```

```
svd(Jnls)$d
```

```
## [1] 1.0051e+07 2.1596e+02 2.2607e+01
```

```
svd(Jatst)$d
```

```
## [1] 1.0000e+07 2.1596e+02 2.2607e+01
```

```
# Even start at the solution?
n0c<-try(nls(form, start=coef(n1), data=data.frame(x=x, y=y), trace=TRUE))
```

```
## 0.018431  (5.39e-08): par = (1.0106 0.89565 0.68027)
```

```
summary(n0c)
```

```
##
## Formula: y ~ 10 * a * (8 + b * log(1 - 0.049 * c * x))
##
## Parameters:
##   Estimate Std. Error  t value Pr(>|t|)
## a 1.01e+00   3.50e-04     2885   <2e-16 ***
## b 8.96e-01   2.26e-03      397   <2e-16 ***
## c 6.80e-01   3.38e-08 20131175   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0513 on 7 degrees of freedom
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 5.39e-08
```

```
## attempts with nlsj
library(nlsj)
```

```
##
## Attaching package: 'nlsj'
```

```
## The following object is masked from 'package:stats':
##
##     numericDeriv
```

```
n0jn<-try(nlsj(form, start=st, data=df, trace=TRUE,control=nlsj.control(derivmeth="numericDeriv")))
```

```
## Warning in nlsj(form, start = st, data = df, trace = TRUE, control =
## nlsj.control(derivmeth = "numericDeriv")): Forcing numericDeriv
```

```
## nlsj: Using default algorithm
## lhs has just the variable  y
##   1 / 2  202.45 :(1  1  0.68027  )  rofftest= 104.8
## fac before = 1  after= 1
## fac= 1  ssnew=0.053966
## <  2 / 3  0.053966 :(1.0106  0.89454  0.68027  )  rofftest= 1.3885
## fac before = 1  after= 1
## fac= 1  ssnew=0.018431
## <  3 / 4  0.018431 :(1.0106  0.89565  0.68027  )  rofftest= 0.00055188
## fac before = 1  after= 1
## fac= 1  ssnew=0.018431
## <  4 / 5  0.018431 :(1.0106  0.89565  0.68027  )  rofftest= 2.9403e-06
```

```
summary(n0jn)
```

```
##
## Formula: y ~ 10 * a * (8 + b * log(1 - 0.049 * c * x))
##
## Parameters:
##   Estimate Std. Error  t value Pr(>|t|)
## a 1.01e+00   3.50e-04     2885   <2e-16 ***
## b 8.96e-01   2.26e-03      397   <2e-16 ***
## c 6.80e-01   3.38e-08 20131174   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0513 on 7 degrees of freedom
## [1] TRUE
## attr(,"cmsg")
## [1] "Termination msg:  Relative offset less than  1e-05 &&"
## attr(,"ctol")
## [1] 2.9403e-06
## attr(,"nres")
## [1] 4
## attr(,"njac")
## [1] 5
```

```
# tmp<-readline("more.")
n0ja<-try(nlsj(form, start=st, data=df, trace=TRUE,control=nlsj.control(derivmeth="default")))
```

```
## nlsj: Using default algorithm
## lhs has just the variable  y
##   1 / 2  202.45 :(1  1  0.68027  )  rofftest= 104.8
## fac before = 1  after= 1
## fac= 1  ssnew=0.053215
## <  2 / 3  0.053215 :(1.0106  0.89454  0.68027  )  rofftest= 1.3738
## fac before = 1  after= 1
## fac= 1  ssnew=0.018431
## <  3 / 4  0.018431 :(1.0106  0.89565  0.68027  )  rofftest= 0.00070916
## fac before = 1  after= 1
## fac= 1  ssnew=0.018431
## <  4 / 5  0.018431 :(1.0106  0.89565  0.68027  )  rofftest= 5.3189e-10
```

```
summary(n0ja)
```

```
##
## Formula: y ~ 10 * a * (8 + b * log(1 - 0.049 * c * x))
##
## Parameters:
##   Estimate Std. Error  t value Pr(>|t|)
## a 1.01e+00   3.50e-04     2885   <2e-16 ***
## b 8.96e-01   2.26e-03      397   <2e-16 ***
## c 6.80e-01   3.40e-08 20024023   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.0513 on 7 degrees of freedom
## [1] TRUE
```

```
## attr(,"cmsg")
## [1] "Termination msg:  Relative offset less than  1e-05 &&"
## attr(,"ctol")
## [1] 5.3189e-10
## attr(,"nres")
## [1] 4
## attr(,"njac")
## [1] 5
```

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 are 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.

Issue: 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`.

- implementation via weights
- implementation via model.frame
- other concerns

Issue: na.action

`na.action` is an argument to the `nls()` function, but it does not appear in obviously in the source code ...

Issue: model frame

`model` is an argument to the `nls()` function, but it does not appear obviously in the source code ...

Issue: documentation of results

We have noticed that there are some important issues relating to the documentation of residuals, fits, and related objects computed by R modeling functions.

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.

Issue: 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. 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.

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

Issue: 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 formulae 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)). 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.

?? TODO: find a way to use getInitial() more easily.

selfStart models in base R

```
SSasymp
SSasympOff
SSasympOrig
SSbiexp
SSfol
SSfpl
SSlogis
SSmicmen
SSgompertz2
SSweibull
```

?? weird output in testing

```
> ls()
[1] "ldata"     "lform"     "lstart2"   "lstartbad" "t"         "y"
> apar <- getInitial(y~SSlogis(t, Asym, xmid, scal), data=ldata)
Error in nls(y ~ 1/(1 + exp((xmid - x)/scal)), data = xy, start = list(xmid = aux[[1L]],  :
  number of iterations exceeded maximum of 50
```

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

$$y\tilde{a} * exp(-b * x) + c * exp(-d * x)$$

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 1s.

Issue: documentation of the results of running nls()

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

?? put in an example and document it.

Concerns with content of the nls 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 (??examples - convergence/termination info, Jsingvals)

Issue: 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 July 2021) we do not have any resolution in mind for this issue.

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

## using a selfStart model
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
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 -- why otherwise?? JN
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)
```

Further comments on partially linear models

Ideally, we believe it would be best to specify all models with a complete set of parameters. That is, the model is as it would be written down for use, rather than for estimation. 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 `algorithm=plinear` setting. Some ideas about a Linear and Nonlinear Discoverer have been proposed by Zhang, Cheng, and Liu (2011), but these seem to relate to an independent (and hence confusing) use of the same term for a class of models that have almost nothing in common with the present issue). We have yet (end of July 2021) to consider how we might translate any ideas to code in an improved `nls()` or similar function.

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 Dr. Heather Turner from John Nash.

```
I'm afraid that I don't know the purpose of the recursive call either.  I know that I wrote the code to
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 th
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 kno
exploring what is happening in the first call as opposed to the recursive call.
```

On Tue, Jun 29, 2021 at 11:50 AM John Nash <Nashjc@uottawa.ca <mailto:Nashjc@uottawa.ca>> wrote:

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.

Cheers, JN

On 2021-06-29 12:33 p.m., Douglas Bates wrote:
> *Attention : courriel externe | external email*
> Thanks for contacting me, John. Can you point me to a file in the gitlab.com <http://gitlab.com>
<http://gitlab.com <http://gitlab.com>> repository that
> contains the definition of setPars?
>
> (By the way, it is probably best to use the email address dmbates@gmail.com <mailto:dmbates@gmail
<mailto:dmbates@gmail.com <mailto:dmbates@gmail.com>> for me. If email
> goes to bates@stat.wisc.edu <mailto:bates@stat.wisc.edu> <mailto:bates@stat.wisc.edu <mailto:bates
it should get forwarded to the gmail.com <http://gmail.com> <http://gmail.com <http://gmail.com>>
> address but sometimes gmail decides that such mail looks suspicious and puts it in the spam folde
why. For
> a long time I used bates@stat.wisc.edu <mailto:bates@stat.wisc.edu> <mailto:bates@stat.wisc.edu

```
<mailto:bates@stat.wisc.edu>> as my "From:" address because it had been my address
> since the 80's but even gmail got suspicious of mail from that address that did not appear to ori
wisc.edu <http://wisc.edu>
> <http://wisc.edu <http://wisc.edu>> domain.)
>
```

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 incumbent on us to provide such material for our own work, and if possible for the existing code.

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

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 questions ???

- do we need a "base" script for each family of test problem, with numbered particular cases?

- how should we document the families and cases? What tags do we need to allow us to quickly select lists of tests
- structure to output the results. AB's draft csv file and runner program.??

??? this is for the quick testing of sets of problems — documented in TestsDoc.

Output of the project

?? see Working Doc etc.

- formal reports
- informal reports
- problem sets
- code and documentation

References

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

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

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, and Duncan Murdoch. 2019. *Nlsr: Functions for Nonlinear Least Squares Solutions*.

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.