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 the 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 "nonlinear," 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: strengths 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 the code has become untidy and overly patched. It is, to our mind, essentially unmaintainable. Moreover, its underlying methods can and should be improved. Let us review some of the issues. We will then propose corrective actions, some of which we have carried out.

Feature: Convergence and termination tests

Within the standard documentation (**man** 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 the 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)
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))
```

4

```
   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` (John C. 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.

Feature: 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 the failure of the simple finite difference approximation of the Jacobian in the `numericDeriv()` function that is a part of `nls()`. `nlsr` can use analytic derivatives, and we 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.)

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 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
```
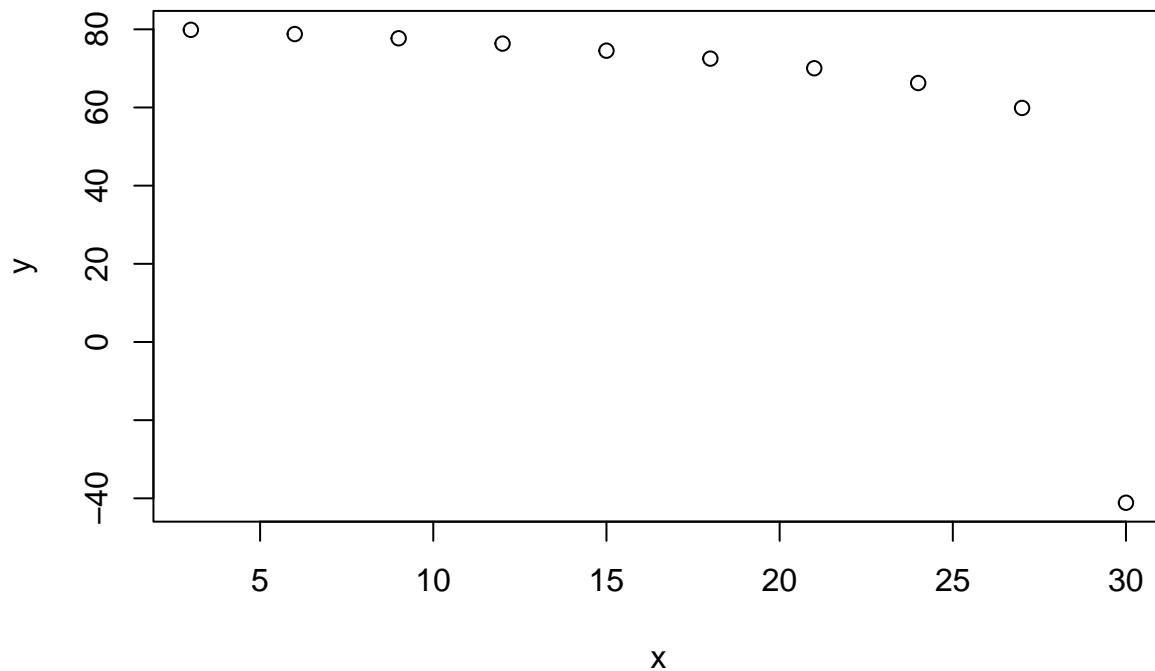


```
st<-c(a=1, b=1,c=c) # set the "default" starting vector
n0<-try(nls(form, start=st, data=df)) # and watch the fun as this fails.
summary(n0)
```

```
##
## Formula: y ~ 10 * a * (8 + b * log(1 - 0.049 * c * x))
##
## Parameters:
##    Estimate Std. Error  t value Pr(>|t|)
## a 1.011e+00  5.327e-04 1.898e+03  < 2e-16 ***
## b 8.998e-01  3.430e-03 2.623e+02 3.09e-15 ***
## c 6.803e-01  5.434e-08 1.252e+07  < 2e-16 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.07803 on 7 degrees of freedom
##
## Number of iterations to convergence: 3
## Achieved convergence tolerance: 1.975e-06
```

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

```
## nlsr object: x
## residual sumsquares =  0.042616  on  10 observations
##     after  4    Jacobian and  5 function evaluations
##   name          coeff          SE      tstat      pval      gradient      JSingval
## a              1.01108      0.0005327      1898  2.976e-21   -5.635e-09      8929755
## b              0.899764     0.00343       262.3  3.091e-15   -1.876e-08        218.5
## c              0.680271     5.461e-08  12456995  5.674e-48   -0.001214         22.6
```

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

```
##           a           b           c
## -1.9261e-11 -1.7963e-10 -4.2633e-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))
```

```
## 0.042616  (5.14e-08): par = (1.0111 0.89976 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   5.33e-04     1898  < 2e-16 ***
## b 9.00e-01   3.43e-03      262  3.1e-15 ***
## c 6.80e-01   5.43e-08 12519388  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.078 on 7 degrees of freedom
##
## Number of iterations to convergence: 0
## Achieved convergence tolerance: 5.14e-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  202.52 :(1  1  0.68027  )  rofftest= 68.929
## [1]   1.1078e-02 -1.0135e-01 -1.6827e-08
```

```
## 1  bdmsk= 1    delta= 0.011078
## 2  bdmsk= 1    delta= -0.10135
## 3  bdmsk= 1    delta= -1.6827e-08
## fac before = 1  after= 1
## fac= 1  ssnew=0.06178
## <  2 / 3  0.06178 :(1.0111  0.89865  0.68027  )  rofftest= 0.6706
## [1]  2.6679e-11  1.1105e-03 -1.9287e-09
## 1  bdmsk= 1    delta= 2.6679e-11
## 2  bdmsk= 1    delta= 0.0011105
## 3  bdmsk= 1    delta= -1.9287e-09
## fac before = 1  after= 1
## fac= 1  ssnew=0.042616
## <  3 / 4  0.042616 :(1.0111  0.89976  0.68027  )  rofftest= 0.00039494
## [1]  6.9450e-12  1.1342e-10 -9.0863e-12
## 1  bdmsk= 1    delta= 6.945e-12
## 2  bdmsk= 1    delta= 1.1342e-10
## 3  bdmsk= 1    delta= -9.0863e-12
## fac before = 1  after= 1
## fac= 1  ssnew=0.042616
## <  4 / 5  0.042616 :(1.0111  0.89976  0.68027  )  rofftest= 1.975e-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   5.33e-04      1898  < 2e-16 ***
## b 9.00e-01   3.43e-03       262  3.1e-15 ***
## c 6.80e-01   5.43e-08  12519388  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.078 on 7 degrees of freedom
## [1] TRUE
## attr(,"cmsg")
## [1] "Termination msg:  Relative offset less than  1e-05 &&"
## attr(,"ctol")
## [1] 1.975e-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  202.52 :(1  1  0.68027  )  rofftest= 68.929
##            a           b           c
##   1.1078e-02 -1.0135e-01 -1.6912e-08
## 1  bdmsk= 1    delta= 0.011078
## 2  bdmsk= 1    delta= -0.10135
```

```
## 3  bdmsk= 1   delta= -1.6912e-08
## fac before = 1  after= 1
## fac= 1  ssnew=0.061985
## <  2 / 3  0.061985 :(1.0111  0.89865  0.68027  )  rofftest= 0.67418
##           a            b            c
## -1.4258e-10  1.1105e-03 -1.8526e-09
## 1  bdmsk= 1   delta= -1.4258e-10
## 2  bdmsk= 1   delta= 0.0011105
## 3  bdmsk= 1   delta= -1.8526e-09
## fac before = 1  after= 1
## fac= 1  ssnew=0.042616
## <  3 / 4  0.042616 :(1.0111  0.89976  0.68027  )  rofftest= 2.6043e-05
##           a            b            c
## -1.8989e-11 -1.5579e-10  6.0451e-13
## 1  bdmsk= 1   delta= -1.8989e-11
## 2  bdmsk= 1   delta= -1.5579e-10
## 3  bdmsk= 1   delta= 6.0451e-13
## fac before = 1  after= 1
## fac= 1  ssnew=0.042616
## <  4 / 5  0.042616 :(1.0111  0.89976  0.68027  )  rofftest= 2.5315e-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   5.33e-04      1898  < 2e-16 ***
## b 9.00e-01   3.43e-03       262  3.1e-15 ***
## c 6.80e-01   5.46e-08  12456994  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.078 on 7 degrees of freedom
## [1] TRUE
## attr(,"cmsg")
## [1] "Termination msg:  Relative offset less than  1e-05 &&"
## attr(,"ctol")
## [1] 2.5315e-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.

Feature: Subsetting

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

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

Feature: na.action

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

Feature: 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 details 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.

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

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. 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.

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 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.

The following example shows how the `SSlogis` selfStart function can generate a set of initial parameters for a 3-parameter logistic curve. The form used by `SSlogis` is

```
y ~ Asym/(1+exp((xmid-tt)/scal))
```

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

```
y ~ b1/(1 + b2*exp(-b3*t))
weed <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443,
          38.558, 50.156, 62.948, 75.995, 91.972)
tt <- 1:12
```

```
NLSformula0 <- y ~ b1/(1+b2*exp(-b3*tt))
NLSformula <- y ~ SSlogis(tt, Asym, xmid, scal)
NLStestdata <- data.frame(y=weed, tt=tt) # should we use standard name?
NLSstart <- getInitial(NLSformula, NLStestdata)
s0 <- getInitial(NLSformula, NLStestdata)
print(s0)
```

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

```
library(nlsj)
s1<-list(b1=s0[1], b2=exp(s0[2]/s0[3]), b3=1/s0[3])
print(as.numeric(s1))
```

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

```
hobblog<-nlsj(NLSformula0, data=NLStestdata, start=s1)
```

```
## Warning in nlsj(NLSformula0, data = NLStestdata, start = s1): Forcing
## numericDeriv
```

```
hobblog
```

```
## Nonlinear regression model
##   model: y ~ b1/(1 + b2 * exp(-b3 * tt))
##    data: NULL
##      b1      b2      b3
## 196.186  49.092   0.314
##  residual sum-of-squares: 2.59
## [1] TRUE
## attr(,"cmsg")
## [1] "Termination msg:  Relative offset less than  1e-05 &&"
## attr(,"ctol")
## [1] 1.1145e-06
## attr(,"nres")
## [1] 1
## attr(,"njac")
## [1] 2
```

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.

selfStart models in base R

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

```
SSasymp        - asymptotic regression model
SSasympOff     - alternate formulation of asymptotic regression model with offset
SSasympOrig    - exponential curve through the origin to an asymptote
SSbiexp        - y ~ ~ A1 * exp(-exp(lrc1)*input) + A2 * exp(-exp(lrc2) * input)
```

```
SSfol              - y ~ Dose * (exp(lKe + lKa - lCl) * (exp(-exp(lKe) * input) -
                            exp(-exp(lKa) * input))/(exp(lKa) - exp(lKe)))
SSfpl              - four parameter logistic model
SSlogis            - three parameter logistic model
SSmicmen           - Michaelis-Menten model for enzyme kinetics
SSgompertz2        - Gompertz model for growth curve data
SSweibull          - Weibull model for growth curve data
```

Strategic issues in selfStart models

Because the Gauss-Newton algorithm is rather unreliable from many starting sets of parameters, selfStart models are more than an accessory to `nls()` but a part of the infrastructure. However, creating such functions is a lot of work, and their documentation (file `./src/library/stats/man/selfStart.Rd`) is quite complicated.

Issue: documentation of the results of running nls()

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

nls() result output according to the documentation

```
A list of:

m              an nlsModel object incorporating the model.

data           the expression that was passed to nls as the data argument. The actual data values
               are present in the environment of the m components, e.g., environment(m$conv).

call           the matched call with several components, notably algorithm.

na.action      the "na.action" attribute (if any) of the model frame.

dataClasses    the "dataClasses" attribute (if any) of the "terms" attribute of the model frame.

model          if model = TRUE, the model frame.

weights        if weights is supplied, the weights.

convInfo       a list with convergence information.

control        the control list used, see the control argument.

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

Example output

To illustrate, let us run the Croucher example.

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

```r
# Try full output version of nls
library(nlspkg) # use the packaged version of nls()
```

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

```
##
## Attaching package: 'nlspkg'
```

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

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

```r
result<-nls(NLSformula, data=NLSdata, start=NLSstart, model=TRUE)
str(result)
```

```
## List of 7
##  $ m          :List of 16
##   ..$ resid     :function ()
##   .. ..- attr(*, "srcref")= 'srcref' int [1:8] 327 15 327 30 22 37 975 975
##   .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##   ..$ fitted    :function ()
##   .. ..- attr(*, "srcref")= 'srcref' int [1:8] 328 16 328 29 23 36 976 976
##   .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##   ..$ formula   :function ()
##   .. ..- attr(*, "srcref")= 'srcref' int [1:8] 329 17 329 31 24 38 977 977
##   .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##   ..$ deviance  :function ()
##   .. ..- attr(*, "srcref")= 'srcref' int [1:8] 330 18 330 31 25 38 978 978
##   .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##   ..$ lhs       :function ()
##   .. ..- attr(*, "srcref")= 'srcref' int [1:8] 331 13 331 26 20 33 979 979
##   .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##   ..$ gradient  :function ()
```

```
##    .. ..- attr(*, "srcref")= 'srcref' int [1:8] 332 18 332 57 25 64 980 980
##    .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##    ..$ conv      :function ()
##    .. ..- attr(*, "srcref")= 'srcref' int [1:8] 333 14 333 34 21 41 981 981
##    .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##    ..$ incr      :function ()
##    .. ..- attr(*, "srcref")= 'srcref' int [1:8] 334 14 334 42 21 49 982 982
##    .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##    ..$ setVarying:function (vary = rep_len(TRUE, np))
##    .. ..- attr(*, "srcref")= 'srcref' int [1:8] 335 20 358 7 27 14 983 1006
##    .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##    ..$ setPars   :function (newPars)
##    .. ..- attr(*, "srcref")= 'srcref' int [1:8] 359 17 366 7 24 14 1007 1014
##    .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##    ..$ getPars   :function ()
##    .. ..- attr(*, "srcref")= 'srcref' int [1:8] 367 17 367 36 24 43 1015 1015
##    .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##    ..$ getAllPars:function ()
##    .. ..- attr(*, "srcref")= 'srcref' int [1:8] 368 20 368 39 27 46 1016 1016
##    .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##    ..$ getEnv    :function ()
##    .. ..- attr(*, "srcref")= 'srcref' int [1:8] 369 16 369 29 23 36 1017 1017
##    .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##    ..$ trace     :function ()
##    .. ..- attr(*, "srcref")= 'srcref' int [1:8] 370 15 376 7 22 14 1018 1024
##    .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##    ..$ Rmat      :function ()
##    .. ..- attr(*, "srcref")= 'srcref' int [1:8] 377 14 377 32 21 39 1025 1025
##    .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##    ..$ predict   :function (newdata = list(), qr = FALSE)
##    .. ..- attr(*, "srcref")= 'srcref' int [1:8] 378 17 379 56 24 56 1026 1027
##    .. .. ..- attr(*, "srcfile")=Classes 'srcfilealias', 'srcfile' <environment: 0x5647a31a3678>
##    ..- attr(*, "class")= chr "nlsModel"
##  $ convInfo   :List of 5
##   ..$ isConv     : logi TRUE
##   ..$ finIter    : int 7
##   ..$ finTol     : num 2.19e-06
##   ..$ stopCode   : int 0
##   ..$ stopMessage: chr "converged"
##  $ data       : symbol NLSdata
##  $ call       : language nls(formula = NLSformula, data = NLSdata, start = NLSstart, model = TRUE,
##  $ dataClasses: Named chr "numeric"
##   ..- attr(*, "names")= chr "xdata"
##  $ model      :List of 2
##   ..$ ydata: num [1:10] 0.699 0.7 0.695 1.039 1.974 ...
##   ..$ xdata: num [1:10] -2 -1.64 -1.33 -0.7 0 0.45 1.2 1.64 2.32 2.9
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language ~ydata + xdata
##   .. .. ..- attr(*, "variables")= language list(ydata, xdata)
##   .. .. ..- attr(*, "factors")= int [1:2, 1:2] 1 0 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "ydata" "xdata"
##   .. .. .. .. ..$ : chr [1:2] "ydata" "xdata"
##   .. .. ..- attr(*, "term.labels")= chr [1:2] "ydata" "xdata"
##   .. .. ..- attr(*, "order")= int [1:2] 1 1
```

```
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 0
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(ydata, xdata)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "numeric"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "ydata" "xdata"
##  $ 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"
```

Concerns with content of the nls result object

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

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

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

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

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

Information that is NOT in the nls result object

`nlsr::nlxb()` solves ostensibly the same problem as `nls()` but only returns

```
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.
```

This is a much smaller result object than the one `nls()` returns. On the other hand, `nlxb` explicitly returns the sum of squares as well as the residual vector and Jacobian. The counts of evaluations are also returned.

Feature: partially linear models and their specification

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

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

```
y ~ a*x + b
```

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

```
y ~ x
```

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

?? comment this!!

Further comments on partially linear models

?? should this be moved to the "Goals" for the future part of paper??

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

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

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

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

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

```
Thanks.

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

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

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

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

The equivalent from minpack.lm is
```

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

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

Feature: indexed parameters

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

```
## The muscle dataset in MASS is from an experiment on muscle
## contraction on 21 animals.  The observed variables are Strip
## (identifier of muscle), Conc (Cacl concentration) and Length
## (resulting length of muscle section).
if(! requireNamespace("MASS", quietly = TRUE)) stop("Need MASS pkg")
mm<- MASS::muscle[1:12,] # take only 1st few values of Strip
str(mm)
```

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

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

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

```
nlev <- nlevels(mm)
withAutoprint({
  ## The non linear model considered is
  ##       Length = alpha + beta*exp(-Conc/theta) + error
  ## where theta is constant but alpha and beta may vary with Strip.
  with(mm, table(Strip)) # 2, 3 or 4 obs per strip
  nl <- nlevels(mm$Strip)
  ## We first use the plinear algorithm to fit an overall model,
  ## ignoring that alpha and beta might vary with Strip.
  musc.1 <- nls(Length ~ cbind(1, exp(-Conc/th)), mm,
                start = list(th = 1), algorithm = "plinear")
  summary(musc.1)

  ## Then we use nls' indexing feature for parameters in non-linear
  ## models to use the conventional algorithm to fit a model in which
  ## alpha and beta vary with Strip.  The starting values are provided
  ## by the previously fitted model.
  ## Note that with indexed parameters, the starting values must be
  ## given in a list (with names):
  ## ?? but why use b here AND in the new formula??
  b <- coef(musc.1)

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

```
## > with(mm, table(Strip))
## Strip
## S01 S02 S03
##   4   4   4
## > nl <- nlevels(mm$Strip)
## > musc.1 <- nls(Length ~ cbind(1, exp(-Conc/th)), mm, start = list(th = 1),
## +     algorithm = "plinear")
## > summary(musc.1)
```

21

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

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

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

Goals of our effort

Here are some of the goals we hope to accomplish.

Code rationalization and documentation

We want

- to provide a packaged version of `nls()` (call it `nlsalt`) coded entirely in R that matches the version in base R or what is packaged in `nlspkg` as described in the "PkgFromRbase" document.

- try to obtain a cleaner structure for the overall `nls()` infrastructure. By this we mean a re-factoring of the routines so they are better suited to maintenance of both the existing `nls()` methods and features as well as the new features we would like to add.

- try to explain what we do, either in comments or separate maintainer documentation. Since we are complaining about the lack of explanatory material for the current code, we feel it is incumbent on us to provide such material for our own work, and if possible for the existing code.

Rationalization of formula specification

?? note automatic detection of linearity. (Birkisson / Driscoll)

```
y ~ x + z + x*z
y ~ a + b*x +c*z + d * (x*z)  # <-- fully specified
y ~ a*Fn1(b, c, x, z)/Fn2()
```

Rationalization of indexed models

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

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

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

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

Provide tests

We need suitable tests in order:

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

A test runner program

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

The project output is available in the Git repository https://gitlab.com/nashjc/improvenls

Formal reports or documentation

- RefactoringNLS.Rms: this document which will become the main report of the project.

- TestsDoc.Rmd: a survey of testing tools in R. ?? need to incorporate MachID and the many tests. ?? Arkajyoti: should we think of working this up into a paper for JSS or the R-Journal? It has more general possibilities and fits into the subject of regression testing, but we would need to provide some literature review. My view is that this COULD be a long term side-interest for your academic work, but that would depend on your own interests as well as opportunities. Note that this document reflects the work in the the "Problem sets and test infrastructure" below.

Informal reports

These are documents used to discuss particular aspects of our work.

- DerivsNLS.Rmd: a document to explain different ways in which Jacobian information is supplied to nonlinear least squares computation in R. File `ExDerivs.R` is intended to provide examples.?? needs documentation??

- ImproveNLS.bib: a consolidated BibTex bibliography for all documents in this project, possibly with wider application to nonlinear least squares in general

- MachineSummary.Rmd: an informal investigation of ways to report the characteristics and identity of machines running tests. `MachID.R` offers a concise summary function to identify a particular computational system used for tests. ?? Note that there remains the get_ram() open issue for Windows 10 in the `benchmarkme` package.

- VarietyInNonlinearLeastSquaresCodes.Rmd: a review of the different algorithms and the many choices in their implementation for nonlinear least squares. ??unfinished but getting there. This is more a JN career interest, and a follow up to the 2014 book.

- PkgFromRbase.Rmd: an explanation of the construction of the `nlspkg` from the code in R-base. We could think of putting this in R-bloggers or similar. Or if we came up with a more general template, then an R-Journal article.

- WorkingDocument4ImproveNLS.Rmd: essentially a way to record what we have worked on. A project diary.

Problem sets and test infrastructure

- ?? we need to make sure these are well organized.

Code and documentation

- nlspkg: a packaged version of the nls() code from R-base. Thanks to Duncan Murdoch

- nlsalt: attempt to mirror nls() behaviour entirely in R. This is UNFINISHED. The effort showed that the structure of the programs was difficult to follow, undocumented, and unsuited to adding improvements like the Marquardt stabilization. We were able to get a pure-R version of numericDeriv() and rework most of the functions of `nlsModel` (but not `nlsModel.plinear`). This work may continue after the project formally ends, but collaboration is likely needed with workers who have a deep knowledge of R internals. `nls-flowchart.txt` was a start at documentation of the structure of `nls()`.

- nls-changes-for-small-residuals-in-nls-R-4.0.2.zip: collected material for the fix by JN to the relative offset convergence criterion failure when there are small residuals in problems sent to `nls()`.

- nlsj: A refactoring of the nls() functionality. `redesign2107.txt` gives some notes.

- nlsralt: a modified version of Nash and Murdoch package `nlsr` with improvements discovered as a result of this project. Need to add subset correctly and document changes e.g., to numerical derivatives. This is for JN rather than AB.

Strategic choices in a nonlinear model estimation

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

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

Opinion for discussion

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

- The functions that compute the residuals and jacobians are frequently computed by presuming the current data and parameters for those functions are available in an environment. As long as the correct environment is used, this provides a surprisingly short syntax to invoke the calculations.

- Weights, subsets, and various contextual controls such as that for the `na.action` that tells our code what to do with undefined or missing numbers. Again, this makes for a very simple invocation of the calculation, but the context is hidden from the user – and it can be difficult for those of us trying to maintain or improve the code to be certain we have the context correct.

- The mixing of R and C code makes for high performance, but without developer documentation leaves programmers with a lot of work to do to adjust the code. We believe in keeping code – at least a

runnable reference version – in a single programming language. If necessary, by measuring ("profiling") the code, we can find bottlenecks and replace just those slower parts of the reference code.

- We believe that a setup, solve, and post-solution structure for complicated functions reduces the number of objects that must be kept in appropriate alignment at any one stage in a set of calculations.

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

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.